COMPUTERBOOKS

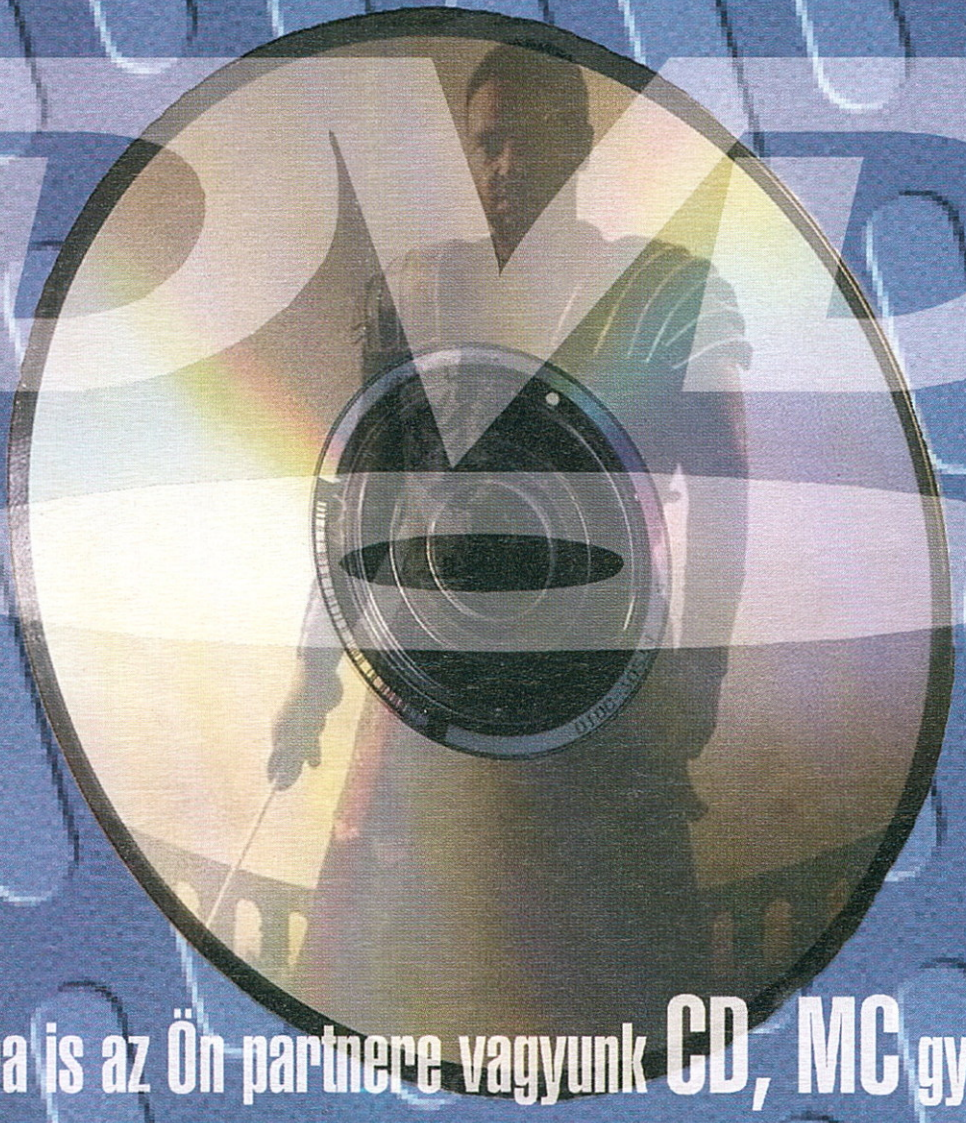
Együtt könnyebb a programozás

# Free Pascal

# MINDENT EGY HELYEN

## masteringtől a sokszorosításig...

## $DVD_5$, $DVD_9$, $DVD_{10}$



## ...továbbra is az Ön partnere vagyunk CD, MC gyártásban.

VTCD VIDEOTON
Kompaktlemez-gyártó Kft.

TÜV Rheinland EUROQUA EN ISO 9001:2000 ZN 75.100.8943

H-8001 Székesfehérvár, Pf.: 175 Tel.: +36-22-533-571, Fax: +36-22-533-599 Email: vtcd@vtcd.hu internet: www.vtcd.hu

Benkő Tiborné
Tóth Bertalan

Együtt könnyebb a programozás

# Free Pascal

CD melléklettel

LEKTOR
Kuzmina Jekatyerina

COMPUTERBOOKS
BUDAPEST, 2006

A könyv készítése során a Kiadó és a Szerzők a legnagyobb gondossággal jártak el. Ennek ellenére hibák előfordulása nem kizárható.
Az ismeretanyag felhasználásának következményeiért sem a Szerzők, sem a Kiadó felelősséget nem vállal.

Minden jog fenntartva. Jelen könyvet vagy annak részleteit a Kiadó engedélye nélkül bármilyen formátumban vagy eszközzel reprodukálni, tárolni és közölni tilos.

©Benkő Tiborné, Tóth Bertalan

Első kiadás, 2006. szeptember

ISBN: 963 618 348 1

©Kiadó: ComputerBooks Kiadó Kft.
1126 Bp., Tartsay Vilmos u. 12.
Telefon: 375-1564; Tel/Fax: 375-3591
E-mail: info@computerbooks.hu
http://www.computerbooks.hu

Felelős kiadó: ComputerBooks Kft. Ügyvezetője
Borítóterv: Székely Edith

Nyomdai munkák: Pressman Bt.

# Tartalomjegyzék

**Előszó** ..... **1**

**Ismerkedés a Free Pascal rendszerrel** ..... **3**
A Free Pascal első látásra ..... 3
Milyen alkalmazásokat fejleszthetünk a Free Pascalban? ..... 4
A Free Pascal Compiler működési módjai ..... 6

**1. A Pascal program szerkezete** ..... **7**
1.1 A legegyszerűbb Pascal program ..... 8
1.2 Egy bonyolultabb Pascal program ..... 8
1.3 Néhány további példa kezdő programozóknak ..... 9

**2. A Pascal nyelv elemei** ..... **11**
2.1 Foglalt szavak és módosítók ..... 11
2.2 A program utasításai és sorai ..... 12
2.3 Azonosítók ..... 12
2.4 Számok ..... 13
2.5 Szövegkonstansok ..... 13

**3. Alaptípusok, deklarációk** ..... **15**
3.1 Változók ..... 15
3.2 A Free Pascal nyelv típusai ..... 16
3.2.1 Egyszerű típusok ..... 16
3.2.1.1 Numerikus adatok tárolása ..... 16
3.2.1.2 Logikai információk tárolása ..... 18
3.2.1.3 Karakteres adatok tárolása ..... 18
3.2.1.4 Sorszámozott típusok ..... 19
3.2.2 Karaktersorozat-típusok ..... 22
3.3 Típusnevek létrehozása, a type deklaráció ..... 24
3.4 Konstansok ..... 25
3.4.1 Normál konstansok ..... 25
3.4.2 Típusos konstansok ..... 25

**4. Alapvető I/O műveletek** ..... **27**
4.1 Írás a képernyőre - a write és a writeln eljárások ..... 27
4.2 Olvasás billentyűzetről - a read és a readln eljárások ..... 31

**5. Operátorok és kifejezések** ..... **37**
5.1 A műveleti jelek csoportosítása ..... 37
5.2 A műveletek elsőbbségi (precedencia) szabályai ..... 39

5.3 Aritmetikai műveletek és kifejezések ..... 40
5.3.1 A precedencia-szabályok aritmetikai kifejezésekben ..... 40
5.3.2 Matematikai függvények használata ..... 42
5.3.3 Típus-átalakítások ..... 43
5.4 Összehasonlító és logikai műveletek ..... 45
5.5 Bitműveletek ..... 48
5.5.1 Bitenkénti logikai műveletek ..... 48
5.5.2 Biteltoló műveletek ..... 50
5.6 Sztringműveletek ..... 52
5.7 Példaprogramok egyszerű számítások elvégzésére ..... 53

**6. A Pascal nyelv utasításai ..... 57**
6.1 Egyszerű utasítások ..... 57
6.1.1 Az üres utasítás ..... 58
6.1.2 Az értékadó utasítás ..... 58
6.1.3 A goto utasítás ..... 58
6.1.4 Eljáráshívás ..... 59
6. 2 Strukturált utasítások ..... 59
6.2.1 Összetett utasítás (blokk) ..... 59
6.2.2 Feltételes utasítások - szelekció ..... 60
6.2.2.1 Az if utasítás ..... 60
6.2.2.2 A case utasítás ..... 65
6.2.3 Ciklusutasítások - iteráció ..... 68
6.2.3.1 A for utasítás ..... 68
6.2.3.2 A while utasítás ..... 72
6.2.3.3 A repeat-until utasítás ..... 74
6.3.2.4 A break és a continue foglalt szavak ..... 76
6.2.4 A with utasítás ..... 78
6.2.5 A kivételkezelés utasításai ..... 78

**7. Strukturált típusok ..... 79**
7.1 Tömbtípusok ..... 79
7.1.1 Művelet azonos típusú tömbökkel ..... 81
7.1.2 Tömbök kezdőértéke ..... 81
7.1.3 Tömbkezelési példaprogramok ..... 82
7.2 Rekordtípusok ..... 85
7.2.1 A rekordok kezelése - a with utasítás ..... 86
7.2.2 Variálható rekord ..... 87
7.2.3 Rekord típusú konstansok ..... 88
7.2.4 Rekordelemű tömbök ..... 89
7.2.5 A rekordok használatát segítő példaprogramok ..... 90
7.3 Halmaztípusok ..... 93
7.3.1 Konstans halmazok ..... 94

7.3.2 Műveletek a halmazokkal ..... 95
7.3.2.1 Halmazok egyesítése, metszete és különbsége ..... 95
7.3.2.2 Halmazkezelő eljárások ..... 95
7.3.2.3 Halmazokra vonatkozó vizsgálatok ..... 96
7.3.3 Halmazkezelési példaprogramok ..... 96
7.4 Állománytípusok ..... 101
7.5 Objektumtípusok ..... 101
7.6 Osztály- és interfésztípusok ..... 101

**8. Eljárások és függvények ..... 103**
8.1 Eljárások és függvények paraméterei ..... 103
8.1.1 Érték szerinti átadott, bemenő paraméterek ..... 103
8.1.2 Cím szerinti átadott be-/kimenő paraméterek ..... 104
8.1.3 Konstans-, bemenő paraméterek ..... 104
8.1.4 Csak Kimenő paraméterek ..... 104
8.2 Függvények ..... 105
8.3 Eljárások ..... 109
8.4 Globális és lokális deklarációk ..... 113
8.4.1 Az azonosítók érvényességi köre, elérhetősége ..... 113
8.4.2 Az azonosítók élettartama ..... 115
8.5 A forward (előzetes) deklaráció ..... 116
8.6 Rekurzív alprogramok ..... 116
8.7 Paraméterek alapértéke ..... 118
8.8 Alprogramok több változatban (overloading) ..... 118
8.9 Operátorok túlterhelése (overloading) ..... 119
8.10 Alprogramok készítése különböző típusú paraméterekkel ..... 122
8.10.1 Egyszerű típusú paraméterek és függvényértékek ..... 123
8.10.2 Tömbtípusú paraméterek és függvényértékek ..... 127
8.10.3 Rekordtípusú paraméterek és függvényértékek ..... 134
8.10.4 Speciális paraméterek és alprogramok ..... 139
8.10.4.1 Típus nélküli paraméterek ..... 140
8.10.4.2 Eljárás- és függvényparaméterek ..... 141
8.10.4.3 Rekurzív alprogramok ..... 142

**9. Programmodulok ..... 149**
9.1 A unit modulok készítése és felhasználása ..... 150
9.1.1 A unitok szerkezete ..... 150
9.1.2 A Free Pascal unitjai ..... 152
9.1.3 Példaprogramok a unit modulok alkalmazására ..... 153
9.2 A library modulok használata ..... 159
9.2.1 Dinamikusan szerkeszthető/betölthető könyvtárak készítése ..... 159
9.2.2 Dinamikusan szerkeszthető/betölthető könyvtárak felhasználása ..... 160
9.2.3 Példaprogram a könyvtármodul kialakítására ..... 161

**10. Dinamikus memóriahasználat ..... 165**
10.1 A mutatótípus és a dinamikus változók ..... 165
10.2 További memóriakezelő függvények és eljárások ..... 168
10.3 Műveletek mutatókkal ..... 169
10.4 Dinamikus tömbök használata ..... 171
10.4.1 Egydimenziós dinamikus tömbök ..... 171
10.4.2 Többdimenziós dinamikus tömbök ..... 173
10.5 Önhivatkozó adatstruktúrák ..... 174
10.6 Feladatok a mutatók használatára ..... 175

**11. Objektum-orientált programozás ..... 193**
11.1 Az egységbezárás ..... 193
11.1.1 Az object típus ..... 193
11.1.2 Az objektumpéldányok létrehozása ..... 195
11.1.2.1 Statikus helyfoglalású objektumpéldányok ..... 195
11.1.2.2 Dinamikus helyfoglalású objektumpéldányok ..... 197
11.1.3 A konstruktor és a destruktor ..... 199
11.1.4 Az adatrejtés elve ..... 202
11.2 Az öröklés ..... 205
11.3 A sokalakúság ..... 210
11.3.1 Az objektumok zártsága - statikus metódusok ..... 210
11.3.2 Virtuális metódusok ..... 211
11.3.3 Absztrakt metódusok ..... 213

**12. Az Object Pascal mód további lehetőségei ..... 215**
12.1 Globális tulajdonságok - a property deklaráció ..... 215
12.2 Objektumosztályok - a class típus ..... 218
12.2.1 Az objektumosztályok (class) felépítése ..... 219
12.2.2 Objektumok létrehozása ..... 222
12.2.3 Öröklés és a sokalakúság ..... 224
12.2.4 Osztályműveletek: is, as ..... 230
12.3 Objektum-interfészek - az interface típus ..... 230
12.4 Kivételek kezelése ..... 234
12.4.1 Kivételek deklarációja ..... 234
12.4.2 Kivételek kiváltása - a raise utasítás ..... 235
12.4.3 A try...except utasítás ..... 236
Kivételek ismételt előidézése ..... 238
Beágyazott kivételek ..... 238
12.4.4 A try... finally utasítás ..... 239
12.4.5 Előre definiált kivételek ..... 240
12.5 Szálak programozása ..... 243
12.5.1 Szálkezelés könyvtári hívásokkal ..... 244
12.5.2 A TThread osztály felhasználása ..... 247

12.6 A Classes modul néhány hasznos osztálya....250

**Elektronikus függelék a CD-mellékleten....261**

**F1. Karaktersorozat-kezelő függvények és eljárások**

**F2. Adatállományok kezelése**

F2.1 Szöveges állományok kezelése

F2.1.1 Szövegfájl deklarálása és megnyitása

F2.1.2 A szöveges állományok I/O műveletei

F2.1.3 A szövegfájl lezárása

F2.1.4 Példák a szöveges fájlok használatára

F2.2 Típusos fájlok használata

F2.2.1 Típusos fájl deklarálása és megnyitása

F2.2.2 Fájlműveletek

F2.2.3 A fájl lezárása

F2.2.4 Példák a típusos állományok használatára

F2.3 Típus nélküli állományok kezelése

F2.4 Állományok törlése és átnevezése

F2.5 Könyvtárak (mappák) kezelése

**F3. Numerikus módszerek programozása**

F3.1 Lineáris egyenletrendszerek megoldása

F3.1.1 Gauss-féle kiküszöbölési eljárás

F3.1.2 Gauss-Jordan módszer

F3.1.3 Fokozatos közelítések módszere (Jacobi-iteráció)

F3.1.4 Gauss-Seidel módszer

F3.1.5 Lineáris egyenletrendszer megoldása LU-dekompozícióval

F3.1.6 Mátrixinvertálás

F3.2 Egyismeretlenes, nemlineáris egyenlet megoldása

F3.2.1 Gyök behatárolása intervallum-felezéssel

F3.2.2 Gyök meghatározása érintő módszerrel (Newton-Raphson módszer)

F3.2.3 Húrmódszer

F3.2.4 A Newton-Raphson és a húrmódszer együttes alkalmazása

F3.2.5 Gyök meghatározása szelőmódszerrel

F3.2.6 Gyök meghatározása fokozatos közelítéssel

F3.3 Interpoláció, regresszió

F3.3.1 Interpoláció

F3.3.2 Lineáris interpoláció

F3.3.3 Lagrange-interpoláció

F3.3.4 Aitken-interpoláció

F3.3.5 Regresszió

F3.4 Numerikus integrálás (numerikus kvadratúra)

F3.4.1 Newton-Cotes kvadratúra módszerek

F3.4.2 Romberg-eljárás
F3.4.3 Nem ekvidisztáns osztású Gauss- és Csebisev-formulák
F3.5 Minimum- és maximumkeresés

**F4. A Free Pascal IDE használata**
F4.1 Könyvtár kiválasztása
F4.2 Új Pascal program írása
F4.3 Meglévő Pascal program futtatása

**Irodalomjegyzék** ..... **263**

**Tárgymutató** ..... **265**

## A CD-melléklet tartalma

| | |
|---|---|
| **\AdobeReader** | Az ***Adobe Reader*** telepítő-állományai. Külön mappák tartalmazzák a ***Windows 2000*** és ***XP***, a ***Windows 98 SE*** valamint a ***Linux*** operációs rendszerek alatt futtatható változatokat. |
| **\Feladatok** | A programozási gyakorlat megszerzését segítő, önálló feldolgozásra szánt több száz feladat, és a feladatmegoldások Pascal nyelvű forrásállományai. |
| **\FreePascal** | A ***Free Pascal Compiler*** (*FPC*) ***2.0.4*** verziójának ***DOS***, ***Win32*** és ***Linux*** (*i386* platformú) változata, valamint a teljes angol nyelvű dokumentációt. |
| **\Info** | A CD-melléklet tartalmával kapcsolatos információk |
| **\Konyv** | Az elektronikus függelék pdf-formátumú állományai, melyek olvasásához szükséges, hogy ***Adobe Reader*** program telepítve legyen a számítógépen (*\AdobeReader* könyvtár vagy *www.adobereader.com*). |
| **\Peldak** | A könyv fejezeteiben bemutatott, kidolgozott példaprogramok forrásállományai, fejezetenkénti csoportosításban. |
| **\Teszt** | A Pascal nyelv önálló tanulását, valamint a tanultak visszakérdezését segítő, ***Windows*** alatt működő elektronikus teszt. |
| **\Telepit** | A könyvhöz tartozó forrásprogramok merevlemezre telepíthető formában |

# Előszó

Évszázadokon át szállóigévé vált a mondás „*Hajózni kell!*". Napjaink emberei nagyjából úgy érezhetik magukat az informatika térhódításától, mint a régi idők emberei az óceánok partján. Akkoriban is megvolt a választás szabadsága – felszállni egy hajóra és elvitetetni magát a világ négy sarkába, vagy megtanulni a hajózás tudományát, és kalandtól hajtva bejárni a tengerek végtelen habjait.

Az informatika óceánjain is rengeteg csónak, hajó, tengerjáró áll rendelkezésünkre az alkalmazások képében, melyeket tudó kezek, emberek készítettek. Ezek a kisebb, nagyobb programok életünk kísérőivé váltak, bár általában nem is gondolunk programként rájuk (mobil telefonálás, Internetezés, számítógépes játékok stb.)

Hihetetlennek tűnhet, azonban az alkalmazások készítéséhez vezető út mindenki előtt nyitva áll. Semmilyen különleges adottság, művészi hajlam nem szükséges ahhoz, hogy programot készítsünk. Jogos tehát a hajdani szólás átfogalmazása „*Programozni kell!*".

Hogyan is kezdjünk hozzá? Valószínűleg van a kezünk ügyében egy számítógép, rajta egy egyszerű szövegszerkesztővel és egy Internet-böngészővel – már írhatjuk is a *javascript/vbscript* utasításokat. Ha a *Microsoft Office* is megtalálható a gépünkön, *VBA* (*Visual Basic for Application*) nyelven hasznos funkciókkal bővíthetjük a *WinWord*, *Excel* stb. alkalmazásokat. A kezdeti sikeres és kevésbé sikeres próbálkozások után valószínűleg szükségét fogjuk érezni egy vezérfonalnak, amely biztonságosan végigvezet bennünket a lehetőségek – végtelennek tűnő – tárházában.

Ez a vezérfonal lehet a **Pascal** nyelv, amit *Niklaus Wirth* 1970-ben, a Zürichi Műszaki Egyetemen, a programozás oktatására dolgozott ki. A nyelv megtanulása azonban nem képzelhető el a gyakorlás lehetősége nélkül, vagyis szükségünk van egy fejlesztőeszközre is. A mai pénzszűke világban az sem hátrány, ha ez az eszköz ingyenes, és többféle hardver és operációs rendszer alatt is használható. A fenti feltételeknek jól megfelel a **Free Pascal** rendszer, ami majdnem teljesen kompatibilis a ***Turbo Pascal 7.0*** és az ***Object Pascal*** nyelvekkel.

A Pascal nyelv választása esetén a könyvünk is jó vezérfonal lehet, amelyben didaktikusan felépített fejezetek sorával, az egyre bonyolultabb nyelvi elemek bemutatásával vezetjük be az *Olvasót* a programozás varázslatos világába.

A könyvünkben követjük a ComputerBooks Kiadó „*Együtt könnyebb a programozás*" sorozatának hagyományait. A fejezetek elméleti részei után, a jobb megértést segítő

példaprogramok állnak. Az első nyolc fejezetben a (Free) Pascal nyelv alapjait ismertetjük, a nyelvi elemektől indulva, az alprogramok használatáig. A következő három passzusban a nagyméretű, a memóriát dinamikusan használó, objektum-orientált programok fejlesztéséhez nyújtunk segítséget. Az utolsó fejezetben összefoglaljuk a grafikus felületű alkalmazások ***Delphi***, ***Kylix*** és ***Lazarus*** rendszerekben történő kialakításához szükséges nyelvi elemeket. A könyvünk itt azonban nem ér véget, mivel a függeléket elektronikus, nyomtatható formában a CD-mellékleten helyeztük el.

Az ingyenes CD-melléklet, a könyvben szereplő példaprogramokon és a függeléken túlmenően, a további gyakorlást és önellenőrzést segítő feladatokat, valamint egy elektronikus tesztet is tartalmaz. Ugyancsak helyet kaptak itt a ***Free Pascal*** fordító ***2.0.4***-es verziójának ***DOS***, ***Windows*** és ***Linux*** alá fejlesztett változatai, valamint a teljes angol nyelvű dokumentáció.

Reményeink szerint, könyvünk a Pascal nyelv tankönyveként is megállja a helyét, segítve a ***Turbo Pascal***, a ***Delphi***, a ***Free Pascal*** és a ***Lazarus*** rendszerekben történő programfejlesztést.

*A szerzők*

**Köszönetnyilvánítás**

Ezúton szeretnénk köszönetet mondani ***Dömötör Dávidnak***, aki azért tanulmányozta a készülő kéziratot, hogy megtanulja a Pascal programozási nyelvet.

Továbbá, köszönjük ***Lehel György*** programozónak a kézirat fejezeteihez fűzött megjegyzéseit.

# Ismerkedés a Free Pascal rendszerrel

A Pascal programozási nyelv történelme immár több mint 30 éves múltra tekint vissza. Ez idő alatt a Pascal nyelv átalakult, kibővült, alkalmazkodott a kor követelményeihez. Talán ennek is köszönhető, hogy napjainkban is több helyen használják, elsősorban a programozás oktatásában.

Az alábbi táblázatban áttekintjük a Pascal szempontjából fontosabb mérföldköveket.

| ***Kihez kötődik?*** | ***Mikor?*** | ***Milyen Pascal?*** | ***Operációs rendszer*** | ***Hardverplatform*** |
|---|---|---|---|---|
| *Niklaus Wirth* | 1970 | *az eredeti Pascal nyelv* | | CDC 6000 |
| | 1980 | *UCSD Pascal* | | Apple II |
| szabvány | 1983 | *ISO 7185 Pascal Standard* | | |
| *Anders Hejlsberg* | 1983 | *Turbo Pascal 1.0* | MS-DOS | IBM PC |
| *Niklaus Wirth, Larry Tesler* | 1985 | *Object Pascal* | Mac OS | Apple Macintosh |
| *Anders Hejlsberg* | 1989 | *Turbo Pascal 5.5* | MS-DOS | IBM PC |
| szabvány | 1990 | *ISO 10206 Extended Pascal Standard* | | |
| szabvány | 1993 | *Object-Oriented Extensions to Pascal Standard* | | |
| *Anders Hejlsberg* | 1992 | *Turbo Pascal 7.0* | MS-DOS | IBM PC |
| *Anders Hejlsberg* | 1995 | *Object Pascal (Delphi)* | Windows | IBM-PC |
| ***Ingyenes!*** | ***1999-től*** | ***Free Pascal*** | ***több*** | ***több*** |

Könyvünkben a fenti táblázat utolsó bejegyzésében szereplő ***Free Pascal*** rendszerrel foglalkozunk, amely fejlesztésekor a *Turbo Pascal 7* és a *Delphi* rendszer *Object Pascal* nyelvével való minél jobb kompatibilitást tűzték ki célul.

### *A Free Pascal első látásra*

A Free Pascal 32/64-bites, többplatformos, ingyenes, nyílt forráskódú, Pascal és Object Pascal fordítóprogram és fejlesztői környezet. A rendszer ingyenességét a *GNU Public License* biztosítja, amely a Pascal nyelven készült fordító és könyvtár forráskódjára is kiterjed. (A *GNU* licencekről bővebben olvashatunk az angol nyelvű *http://www.gnu.org/licenses/licenses.html* oldalon.)

Az aktuális, könyvünk készítéséhez is felhasznált 2.0.4-es verzió, az alábbi (hardver- és szoftver-) platformokat támogatja:

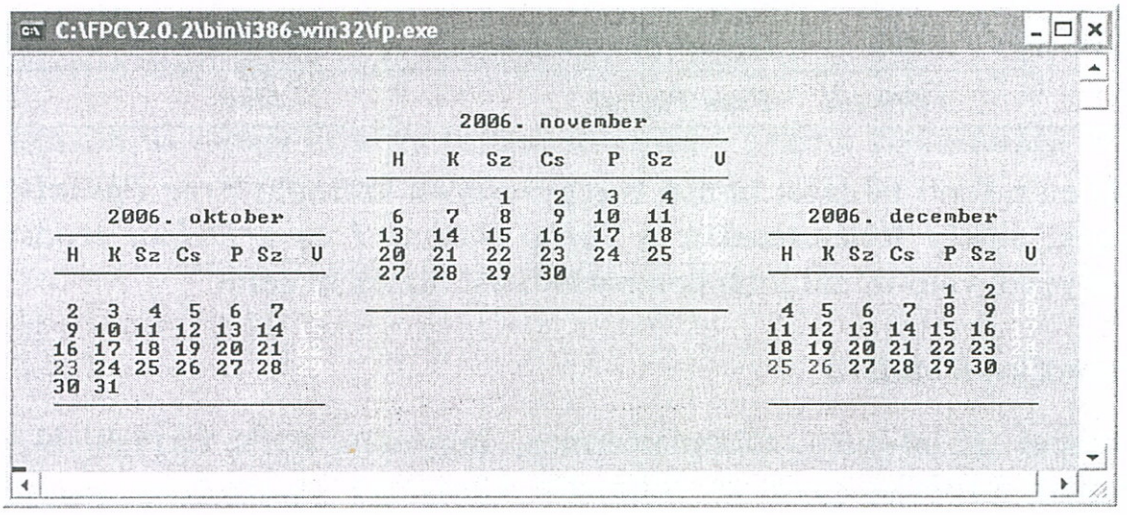
| *Processzorok:* | *Operációs rendszerek:* |
|---|---|
| iA-32: Intel 80386 és kompatibilis,<br>AMD64: x86_64,<br>PowerPC,<br>ARM,<br>Sparc v8 és v9. | Linux: minden CPU-ra ,<br>FreeBSD,<br>Mac OS X és Darwin (PowerPC),<br>Mac OS,<br>DOS; Go32V2 extender,<br>Win32,<br>OS/2: EMX és natív,<br>Novell NetWare. |

A Free Pascal legutóbbi verzióját magunk is letölthetjük a *http://www.freepascal.org/* honlapról, azonban a ***Free Pascal Compiler*** *(FPC)* ***2.0.4*** verziójának ***DOS***, ***Win32*** és ***Linux*** *(i386* platformú) változatát, valamint a teljes angol nyelvű dokumentációt a könyvünk CD-mellékletén is elhelyeztük.

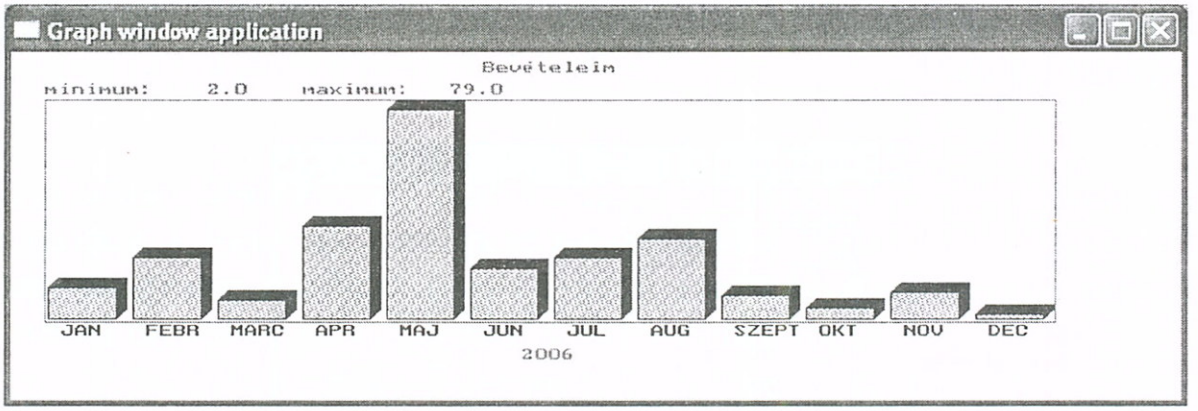
### *Milyen alkalmazásokat fejleszthetünk a Free Pascalban?*

A ***Free Pascal*** önmagában 32/64-bites, szöveges felhasználói felületű Pascal nyelvű programok fejlesztését teszi lehetővé, azonban a grafikus könyvtárak alkalmazásával grafikát is megjeleníthetünk segítségével. Az ***FPC*** szöveges és grafikus lehetőségei a ***Turbo Pascal 7*** képességeire korlátozódnak. Maga a fejlesztői környezet is szöveges felülettel rendelkezik.

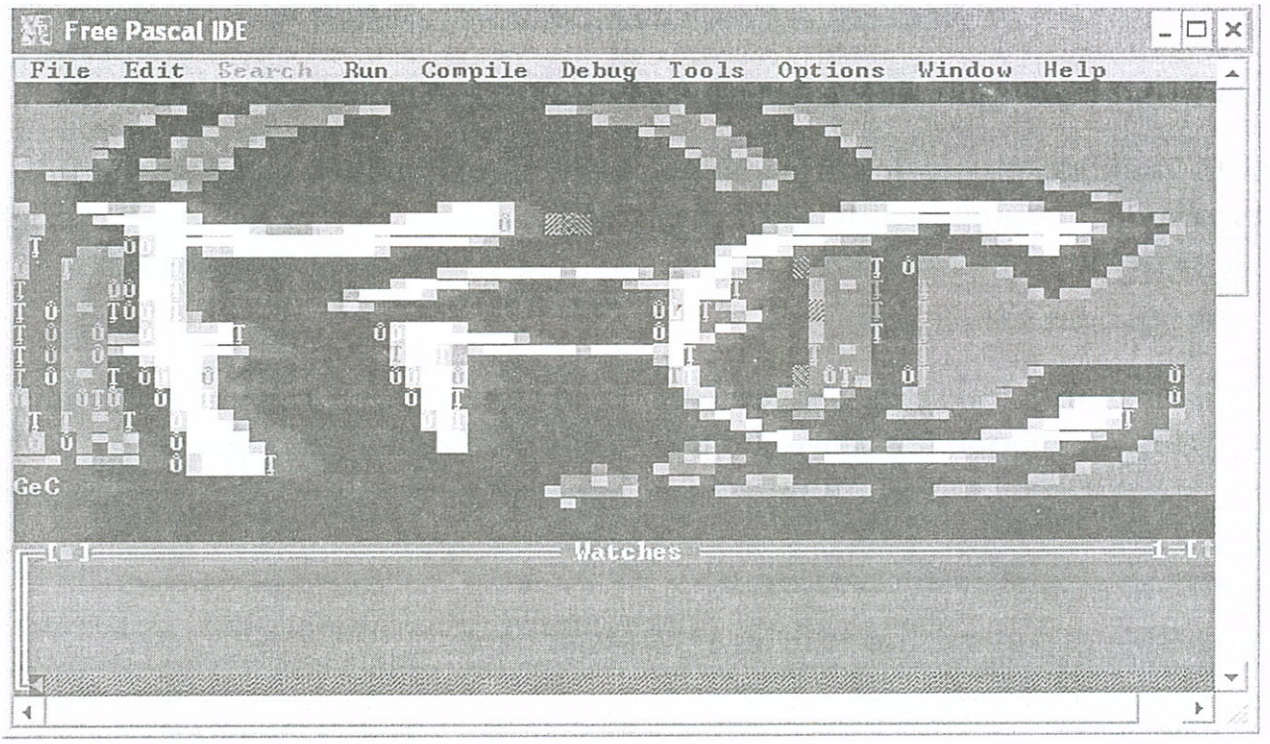
A szöveges felületű Pascal program futásának eredménye:




Grafikus (***Graph*** unit) programok futtatása során a szokásos szöveges ablak mellett egy másik ablak is megjelenik, amelyik a grafikát tartalmazza. Ez az ablak mindig fekete hátterű, ezért könyvünkben átszíneztük a hátterét:



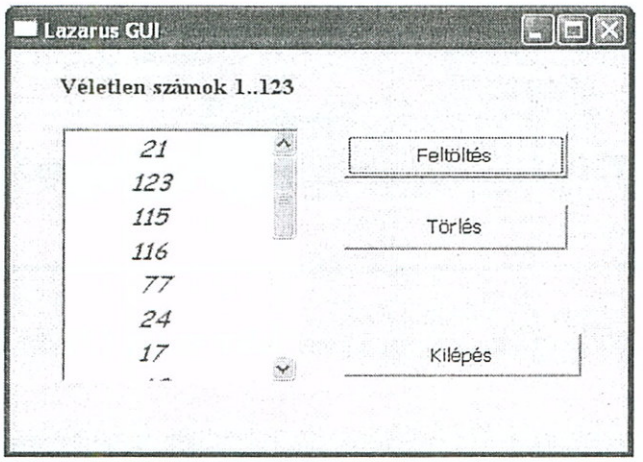

Végezetül vessünk egy pillantást a fejlesztői környezet (*Free Pascal IDE*) bejelentkező ablakára!




Természetesen korunk igényei messze meghaladják a fentiekben vázolt lehetőségeket, azonban a Pascal nyelv elsajátítása után tovább is léphetünk. A ***Delphi*** rendszerben megszokott vizuális, gyors alkalmazásfejlesztés könyvtárait és fejlesztői felületét egy másik project, a ***Lazarus*** tartalmazza. A ***Lazarus***, a *Free Pascal* fordítóra épülve valósítja meg a *Delphi* rendszer emulációját. A ***Lazarus*** szintén ingyenesen letölthető a *http://www.lazarus.freepascal.org/* weboldalról.

A ***Lazarus*** rendszerrel könyvünkben nem foglalkozunk, azonban az utolsó fejezetben áttekintjük a működéséhez szükséges (*ObjFpc*) nyelvi elemeket.

A ***Lazarus*** környezetben készített grafikus felületű alkalmazás (*Graphical User Interface*, *GUI*) ablaka:




### *A Free Pascal Compiler működési módjai*

A fordító működését ún. fordítói előírásokkal, direktívákkal szabályozhatjuk. Most csak a működési módok beállításával, és az egyes módok jellegzetességeivel foglalkozunk. A módot a programmodulok elején a ***$mode*** direktívával állíthatjuk be:

```
{$mode működési_mód}
```

Amennyiben nem adjuk meg a direktívát, az alapértelmezés szerinti ***FPC*** mód érvényesül. Az egyes módok legfontosabb jellemzőit táblázatban összegeztük.

| *Néhány összehasonlítási szempont* | *FPC* | *TP* | *Delphi* | *OBJFPC* |
|---|---|---|---|---|
| *@alprogram* az alprogram-értékadásoknál, | √ | — | — | √ |
| a **forward** deklaráció és a definíció teljes egyezősége, | √ | — | — | √ |
| *overloading* azonos alprogramnevek esetén, | √ | — | — | √ |
| egymásba ágyazott megjegyzések, | √ | — | — | √ |
| az ***ObjPas*** modul betöltődik a ***System*** unitot követően, | — | — | √ | √ |
| a ***pchar*** szöveg automatikusan **string**gé alakul, | √ | — | — | √ |
| adott **class** osztályon belül, a metódusparaméterek nevei nem egyezhetnek meg a tulajdonságnevekkel. | ⊗ | ⊗ | — | √ |

Az egyes módok jelentését szintén táblázatban rögzítettük:

| | |
|---|---|
| ***FPC*** | a *Free Pascal Compiler* alapértelmezett módja, |
| ***TP*** | *Turbo Pascal* mód, |
| ***Delphi*** | *Delphi* mód, |
| ***OBJFPC*** | *Object Free Pascal Compiler* mód. |

# 1. A Pascal program szerkezete

A Pascal alkalmazások egy vagy több modulból épülnek fel. Könyvünk elején csak az egyetlen modult, a főmodult tartalmazó programokkal foglalkozunk. A főmodulban a főprogram (**program**) található, amely kijelöli a lefordított alkalmazás belépési pontját. (A további modulokkal, a unitokkal a 9. fejezetben foglalkozunk.)

A Pascal program általános felépítését megjegyzésekkel ellátva mutatjuk be, majd pedig az egyes részek használatával példák segítségével ismerkedünk meg.

*A Pascal program általános szerkezete:*

```
{ programfej }
    { globális hatású fordítási direktívák }
    program programnév;

{ deklarációs rész }
    uses  { a használt könyvtárak (modulok) felsorolása };
    label { címkék deklarációja };
    const { konstansok deklarálása };
    type  { típusok deklarálása };
    var   { változók deklarálása };

    procedure eljárásnév (paraméterek);
    { lokális deklarációk }
    begin
      { az eljárás törzsének utasításai }
    end;

    function függvénynév (paraméterek): típus;
    { lokális deklarációk }
    begin
      { a függvény törzsének utasításai }
      függvénynév := utasítás; { a függvény értéke }
    end;

{ programtörzs }
    begin
      { lokális hatású fordítási direktívák }
      { a főprogram blokkjának utasításai }
    end.
```

A fenti szerkezetből jól látható a Pascal nyelven írt programok három fő része:

- *Programfej*: a Pascal program feje a **program** foglalt szóból és a programnévből áll.
- *Deklarációs rész*: ahol a programban használni kívánt saját nevek deklarációi helyezkednek el.

- *Programtörzs*: A legegyszerűbb Pascal programokban a deklarációs rész hiányozhat, azonban a program törzsét mindig meg kell adni. A programtörzs a **begin** és az **end**. közötti végrehajtható utasításokat tartalmaz.

## 1.1 A legegyszerűbb Pascal program

```
program legegyszerubb;
begin
 // végrehajtható utasítások
end.
```

Megjegyzések a fenti programmal kapcsolatosan:

- A programnévnek betűvel kell kezdődnie, amelyet betű, számjegy vagy aláhúzás jel (_) követhet. A programfejet pontosvessző (;) zárja.
- A **begin** a főprogram blokkjának kezdő utasítás-zárójele, amelyet végrehajtható utasítások követnek. A **begin** egyben kijelöli lefordított program belépési pontját.
- A főprogram blokkját záró **end** utasítás-zárójel után a pontjel (.) áll.
- A lemezen tárolt Pascal program kiterjesztése *.PAS*. Nem szükséges, hogy a forrásállomány neve megegyezzen a program nevével, azonban azonosak is lehetnek.

## 1.2 Egy bonyolultabb Pascal program

```
{ Két egész szám összegének kiszámolása }
program bonyolultabb;

var
    a, b, c: integer;  // tárolók deklarálása

begin
   { az egész számok beolvasása: }
   write('Kérem az első egész számot  : '); // szöveg megjelenítése
   readln(a);                               // egész szám beolvasása
   write('Kérem a második egész számot: ');
   readln(b);
   { az összegzés elvégzése,
     és az eredmény tárolása a c tárolóban: }
   c := a + b;
   // az eredmény megjelenítése
   writeln('A két egész szám összege: ', c:6);
   // várakozás az Enter billentyű leütésére
   readln;
end.
```

*A program futásának eredménye:*

```
Kérem az első egész számot  : 12
Kérem a második egész számot: 23
A két egész szám összege:     35
```

A „bonyolultabb" program deklarációs része nem üres, hanem három, egész számok tárolásra alkalmas programelemet, ún. változót definiál. A programtörzsben beolvassuk a számításhoz szükséges adatokat, elvégezzük az összegzést, és megjelenítjük az eredményeket. A billentyűzetről való beolvasáshoz a ***readln***, míg a képernyőn való megjelenítéshez a ***write*** és a ***writeln*** eljárásokat használjuk. Az összegzés eredményét az értékadó utasítással (:=) a *c* tárolóba töltjük. Az összegzést a matematikából jól ismert + műveleti jellel (operátorral) írtuk elő.

A példaprogram megértését megjegyzésekkel segítettük. Free Pascal programban egy- és többsoros magyarázatokat egyaránt elhelyezhetünk, három lehetőség közül is választhatunk:

```
// egysoros megjegyzés, amely a sor végéig tart
{ egysoros megjegyzés, amely a záró kapcsos zárójelig tart }
(* egysoros megjegyzés, amely a * és a záró kerek zárójelig tart *)

// Többsoros
// megjegyzés

{  Többsoros
   megjegyzés }

(* Többsoros
   megjegyzés *)
```

## 1.3 Néhány további példa kezdő programozóknak

Az alábbi példaprogramokkal a Pascal nyelvvel most ismerkedő Olvasó első lépéseit kívánjuk segíteni. A Pascal nyelvet már alap szinten ismerő Olvasó nyugodtan lapozhat a következő fejezetre.

Írjunk programot, amely szöveget ír ki a képernyőre és a szöveg kiírása után sort emel! (*elso*)

A program listája:

```
program elso;
begin
  writeln('Pascal nyelvű program írása.');
  readln;
end.
```

*A program futásának eredménye:*

```
Pascal nyelvű program írása.
```

Megjegyzések a programhoz:

- A ***writeln*** eljárással az aposztrófok között megadott szöveget a képernyőre írjuk, és a kiírást követően sort emelünk – ezt jelzi az eljárás nevében az ***ln*** (*line*).

- A Pascal utasításokat pontosvesszővel zárjuk. Felhívjuk a figyelmet arra, hogy a programsor vége nem minden esetben jelenti az utasítás végét.
- A paraméter nélküli ***readln*** eljárást arra használjuk, hogy a program futásakor, a szöveges ablakban megjelenő eredményt addig lehessen látni, míg *Enter*t nem ütünk a billentyűzeten. Az *Enter* leütésével kilépünk a programból, mivel a ***readln*** az utolsó végrehajtható utasítás.
- A programunkat az *elso.pas* állomány tartalmazza.

Írjunk programot, amely szöveget jelenít meg a **write** és a **writeln** eljárások felhasználásával! A programot lássuk el megjegyzésekkel! (*masodik*)

```
program masodik;
begin
  write('Pascal ');     // Az első szó kiírása
  write('nyelvű ');     // a második szó kiírása
  writeln('program írása.');  { a szöveg után
                                (* soremelés *)
                              }
  readln; (* Felfüggeszti a program futását és
             { az Enter }
             billentyű leütésére vár
          *)
end.
```

*A program futásának eredménye:*

```
Pascal nyelvű program írása.
```

A ***write*** eljárás megjeleníti az aposztrófok között megadott szöveget, azonban ezt követően nem emel sort. A (* és *), valamint a { és } jelek között többsoros megjegyzéseket is megadhatunk. A magyarázatok egymásba skatulyázásakor különböző típusú megjegyzésjeleket kell használnunk.

Készítsünk programot, amely több sorban jelenít meg szöveget a ***write*** és a ***writeln*** eljárások segítségével! (*harmadik*)

```
program harmadik;
begin
  write('Pascal nyelvű program');
  writeln;                          // soremelés
  writeln('       írása.');
  readln;
end.
```

*A program futásának eredménye:*

```
Pascal nyelvű program
       írása.
```

# 2. A Pascal nyelv elemei

A Pascal nyelv jelkészlete az ***ASCII*** karakterkészlet elemeiből épül fel. A különböző Free Pascal platformok ugyan kibővítik ezt a karakterkészletet (***ANSI***, ***Unicode*** stb.), azonban a bővített karaktereket csak megjegyzésekben és szövegkonstansokban használhatjuk.

## 2.1 Foglalt szavak és módosítók

A Pascal nyelv utasításaiban és deklarációiban használt kulcsszavakat foglalt szavaknak nevezzük. A foglalt szavakat a szerint csoportosítjuk, hogy melyik nyelvből vette át a Free Pascal. A fordító ***TP*** módban csak a *Turbo Pascal* foglalt szavait ismeri fel. Ezzel ellentétben a *Delphi* foglalt szavait alapértelmezés szerint értelmezi a fordító.

*A Turbo Pascal foglalt szavai:*

| | | | |
|---|---|---|---|
| absolute | else | nil | set |
| and | end | not | shl |
| array | file | object | shr |
| asm | for | of | string |
| begin | function | on | then |
| break | goto | operator | to |
| case | if | or | type |
| const | implementation | packed | unit |
| constructor | in | procedure | until |
| continue | inherited | program | uses |
| destructor | inline | record | var |
| div | interface | reintroduce | while |
| do | label | repeat | with |
| downto | mod | self | xor |

*A Delphi és a Free Pascal további foglalt szavai:*

| *Delphi foglalt szavak* | | *Free Pascal foglalt szavak* |
|---|---|---|
| as | library | dispose |
| class | on | exit |
| except | property | false |
| exports | raise | new |
| finalization | threadvar | true |
| finally | try | |
| initialization | | |
| is | | |

A Free Pascal további szavakat is használ, amiket módosítóknak nevezünk. Ezek nem foglalt szavak, vagyis saját azonosítókként is használhatjuk őket, azonban bizonyos szövegkörnyezetben a fordító számára speciális jelentést hordoznak.

*A Free Pascal nyelv módosítói:*

| | | | |
|---|---|---|---|
| **absolute** | **external** | **override** | **safecall** |
| **abstract** | **far** | **pascal** | **softfloat** |
| **alias** | **far16** | **private** | **stdcall** |
| **assembler** | **forward** | **protected** | **virtual** |
| **cdecl** | **index** | **public** | **write** |
| **cppdecl** | **name** | **published** | |
| **default** | **near** | **read** | |
| **export** | **oldfpccall** | **register** | |

Megjegyezzük, hogy a Free Pascal nyelv típusai és konstansai a ***System*** modulban definiált azonosítók.

## 2.2 A program utasításai és sorai

A Pascal nyelvben a program utasításait pontosvesszővel (;) zárjuk. A programot záró **end** utasítás kivételt képez, amely után pontot (.) kell tenni. Egy sorban több utasítást is elhelyezhetünk, illetve egy utasítást több sorban is megadhatunk.

## 2.3 Azonosítók

A Pascal programban minden általunk létrehozott elemnek (változónak, konstansnak, alprogramnak stb.) nevet kell adni, hogy hivatkozni tudjunk rájuk. A névnek egyedinek kell lennie; hogy egyértelműen azonosítsa az elemet. Természetesen az azonosító nem lehet foglalt szó.

Az azonosítónak betűvel vagy aláhúzás jellel (_) kell kezdődnie, és nem tartalmazhat szóközt. Az első karakter után további betűk, számjegyek és aláhúzás jelek következhetnek. A Free Pascal az azonosítók első 255 karakterét különbözteti meg.

Nézzünk néhány példát a helyes azonosítókra!

```
x   y1   w1a   oldal   tav1   a_oldal   KorKerulet   iSzamlalo
```

Az alábbi azonosítók azonban hibásak, mivel meg nem engedett karaktert tartalmaznak:

```
x%   y+1   w-a   7oldal   tav_*   a oldal
```

A Pascal nyelv nem különbözteti meg a kis- és a nagybetűket az azonosítókban (a foglalt szavakban sem), így az alábbi nevek ugyanazt az azonosítót jelölik:

```
kor_terulet   Kor_terulet   Kor_Terulet   KOR_TERULET
```

## 2.4 Számok

A Pascal nyelvben egész és valós számokat használunk. Az egész számok megadhatók előjellel vagy előjel nélkül (a pozitív előjel elhagyható):

```
1   -12   +52
```

Az egész értékeket decimális, hexadecimális ($), oktális (&) és bináris (%) számrendszerben egyaránt is megadhatjuk. (***Delphi*** és ***TP*** módban azonban csak a decimális és hexadecimális forma használható.)

```
$0      $FFFFFF    $123456789ABCDEF
&1223   &1017      &1234567
%101    %1111111   %1011000111
```

A valós számokat kétféleképpen is megadhatjuk.

- Tizedes törtként:

```
121.56   -32.789   0.0023
```

- Hatványkitevős alakban, ahol az *e* vagy az *E* exponensjel után álló egész szám 10 hatványkitevőjét jelöli.

```
2.3124E+2   -7.89e-5   2.1E-3
```

*Megjegyzés a valós számokkal kapcsolatban:*

- Tizedes tört esetén az egész részt és a tört részt is mindig meg kell adnunk:

```
1.23   0.7912   0.0023   1.0
```

- Tizedes pontot és nem tizedes vesszőt használunk.

## 2.5 Szövegkonstansok

A Pascal nyelvben a szövegkonstanst egyszeres idézőjelek, aposztrófok ('') közé kell tenni:

```
'szövegkonstans megadása'
```

Amennyiben a szövegen belül aposztróf szerepel, akkor azt megkétszerezve ('') kell a szövegkonstansban elhelyezni:

```
'a szöveget aposztrófok '''' közé kell tenni'
```

A szövegmegadás további lehetőségeivel a **string** típust ismertető 3. fejezetben foglalkozunk.

# 3. Alaptípusok, deklarációk

A Pascal erősen típusos programozási nyelv. Ez a megállapítás azt jelenti, hogy a fordító mindig figyel arra, hogy csak megfelelő adatokkal dolgozzunk. Ehhez az információit a különböző nyelvi elemekhez rendelt típusokból nyeri. A nyelvi elemek és típusok összerendelése a deklaráció.

## 3.1 Változók

A Pascal nyelvű programokban a használni kívánt változókat deklarálnunk kell, megadva ezzel a fordítónak a változók tulajdonságait.

*A változók tulajdonságai:*

- A programban az adatokat változókban tároljuk, melyek tartalma módosítható.
- A változó adattípusa meghatározza, hogy egy változó milyen értéket vehet fel, és milyen műveleteket végezhetünk rajta.
- A változó számára a fordítóprogram a típusnak megfelelően egy vagy több bájtot foglal le a memóriában.
- A tárterület tartalmazza a változó *aktuális értékét*, amely a program futása során változhat.
- A programban a változókra névvel hivatkozunk, ezért minden változót azonosítóval kell ellátnunk.

A Pascal programban minden változót (de csakis egyszer) deklarálni kell a program deklarációs részében a **var** foglalt szó után.

```
var
  adat1, adat2: integer;
  eredmeny: real;
```

A deklarációban a változó nevét kettősponttal elválasztva követi a változó típusa, amely után pontosvessző áll. Ha egy típushoz több változó tartozik, akkor azok neveit vesszővel elválasztva kell felsorolni.

Lehetőség van arra is, hogy a változóknak a deklarációban kezdőértéket adjunk:

```
var
   adat12 : integer = 123;
```

A változók értékét a **begin**...**end**. blokkban; az értékadó utasításban módosíthatjuk, az értékadás jobb oldalán szerepeltetve a változó nevét:

```
begin
   adat12 := 0;
end.
```

## 3.2 A Free Pascal nyelv típusai

A Free Pascal nyelv sokféle beépített típussal rendelkezik, melyeket az alábbiak szerint csoportosíthatjuk.

- *Egyszerű típusok: sorszámozott* (egész, karakteres, logikai, felsorolt és résztartomány), valamint a *valós* típusok,
- *Karaktersorozat típusok:* rövid és hosszú, nullavégű és széles sztringek,
- *Strukturált típusok:* tömb, rekord, halmaz, állomány, objektum, osztály, osztályreferencia és interfész,
- *Mutatótípus,*
- *Eljárástípus,*
- *A variáns (variant) típus.*

Ebben a fejezetben csak az egyszerű és a karaktersorozat típusokkal foglalkozunk. A további típusokat könyvünk későbbi fejezeteiben dolgozzuk fel.

### 3.2.1 Egyszerű típusok

Az egyszerű típusú változók numerikus vagy karakteres értékek tárolására alkalmasak. Először ismerkedjünk meg a numerikus értékek tárolását biztosító típusokkal! A típusokat az értékkészletükkel, illetve a memóriaigényükkel jellemezhetjük.

Megjegyezzük, hogy a ***sizeof*** függvény segítségével magunk is lekérdezhetjük egy adott típus vagy változó tárolásához szükséges memóriaterület bájtokban kifejezett méretét:

#### *3.2.1.1 Numerikus adatok tárolása*

Az ***egész*** adatok tárolására 1- ,2- ,3- vagy 8-bájtos típusok állnak rendelkezésünkre:

| Típus | Értékkészlet | Helyfoglalás |
|---|---|---|
| ***shortint*** | `-128..127` | előjeles 8 bit (1 bájt) |
| ***smallint*** | `-32768..32767` | előjeles 16 bit (2 bájt) |
| ***longint*** | `-2147483648..2147483647` | előjeles 32 bit (4 bájt) |
| ***int64*** | `-9223372036854775808..9223372036854775807` | előjeles 64 bit (8 bájt) |
| ***integer*** | ***smallint, shortint*** vagy ***int64*** | előjeles 2,4 vagy 8 bájt |
| ***byte*** | `0..255` | előjel nélküli 8 bit (1 bájt) |
| ***word*** | `0..65535` | előjel nélküli 16 bit (2 bájt) |
| ***longword*** | `0..4294967295` | előjel nélküli 32 bit (4 bájt) |
| ***qword*** | `0..18446744073709551615` | előjel nélküli 64 bit (8 bájt) |
| ***cardinal*** | ***word, longword*** vagy ***qword*** | előjel nélküli 2,4 vagy 8 bájt |

Az ***integer*** és a ***cardinal*** általános egész típusok mérete függ a fejlesztői rendszertől (16-bites, 32-bites stb.). Az alapértelmezett ***FPC*** és a ***TP*** módokban az ***integer*** típus 16-bites, míg ***ObjFPC*** és ***Delphi*** módokban 32-bites. Ezzel szemben a ***cardinal*** típus mind a négy módban 32-bites.

A Free Pascal programokban az egész számok tárolására leggyakrabban az ***integer*** típust használjuk, amennyiben nem elég a számításokra az értékkészlet, a ***longint*** típust is választjuk. A legnagyobb ***integer*** típusú egész számot (32767) a ***MAXINT*** konstans, a legnagyobb ***longint*** egész számot (2147483647) pedig a ***MAXLONGINT*** konstans tárolja, melyeket a programunkban is felhasználhatjuk:

```
program egesz;
var
  i, j, max: integer;
  k, lmax: longint;
  n : int64;
begin
  i := 12;
  j := 12345;
  max := MAXINT;
  k := 213291972;
  lmax := MAXLONGINT;
  n := MAXLONGINT * MAXINT;
end.
```

A ***valós*** adatok tárolására szintén több típus közül választhatunk:

| Típus | Értékkészlet | Pontosság | Helyfoglalás |
|---|---|---|---|
| ***real*** | ***single*** vagy ***double*** | ? | 4 vagy 8 bájt |
| ***single*** | $1.5 \cdot 10^{-45}$ .. $3.4 \cdot 10^{38}$ | 7 - 8 | 4 bájt |
| ***double*** | $5.0 \cdot 10^{-324}$ .. $1.7 \cdot 10^{308}$ | 15 - 16 | 8 bájt |
| ***extended*** | $3.4 \cdot 10^{-4932}$ .. $1.1 \cdot 10^{4932}$ | 19 - 20 | 10 bájt |
| ***comp*** | $-2^{-63}+1$ .. $2^{63}-1$ | 19 - 20 | 8 bájt |
| ***currency*** | -922337203685477.5808.. 922337203685477.5807 | 19 - 20 | 8 bájt |

*Megjegyzés*: Az értékkészletet abszolút értékben adtuk meg, ugyanez vonatkozik a negatív számokra is, de ellenkező előjellel. A Free Pascal csak az IEEE szabvány szerinti lebegőpontos típusokat támogatja. Amennyiben a platform nem ismeri ezeket a típusokat, úgy a Free Pascal emulálja azokat.

A programjainkban a valós számok tárolására általában a ***real*** típust használjuk. Az Intel 80x86 alapú platformokon a ***real*** típus a ***double*** típust jelöli.

A ***currency*** típus szintén valós számot tárol, azonban fixpontos formában. Az adat 64-bites egész számként tárolódik, és mindig négy tizedes jegyet tartalmaz. (A tárolt

szám tehát a négy tizedes pozíciót tartalmazó valós szám tízezerszerese). Az adattípus bevezetésének célja, hogy pénzügyi számításoknál a kerekítési hibák minimálisak legyenek.

Nézzünk néhány példát a valós típusú változók deklarációjára!

```
program valos;
var
  a, b: real;
begin
  a := 12.23;
  b := 1.22e3;
end.
```

### *3.2.1.2 Logikai információk tárolása*

A logikai típusú változókat logikai értékek tárolására és logikai műveletek végzésére használjuk. A logikai változók csak kétféle értéket vehetnek fel: **true** (igaz) és **false** (hamis). A szabványos Pascal csak a ***Boolean*** logikai típust tartalmazza. A Free Pascalban azonban – más nyelvekkel való kompatibilitás érdekében – újabb típusok is megtalálhatók:

| *Típusazonosító* | *Értékkészlet* | *Méret* |
|---|---|---|
| *Boolean* | `false..true` | 8-bites |
| *ByteBool* | `false..true` | 8-bites |
| *WordBool* | `false..true` | 16-bites |
| *LongBool* | `false..true` | 32-bites |

```
program bool;
var
  van, hiba: boolean;
begin
  van := true;
  hiba := false;
end.
```

### *3.2.1.3 Karakteres adatok tárolása*

Az egybájtos **char** típusú változók csak egyetlenegy karakter tárolására alkalmasak. A Free Pascalban kétféle karakterkódolásnak megfelelően használhatunk típusokat, a 8-bites ***AnsiChar*** és a 16-bites ***WideChar*** (***WChar***) típusokat. A Pascal nyelv szabványos ***char*** típusa megegyezik az ***AnsiChar*** típussal.

A ***char*** típus sorszámozott típus. Minden karakterhez tartozik egy sorszám (a karakter kódja: 0..255), amely kijelöli a karakter helyét az ASCII kódtáblában.

A karakteres konstansokat egyszeres aposztrófok között kell megadnunk, azonban használhatjuk a karakterkódot is a # jel után:

```
'*'   '0'   'N'   #78   #$4E
```

```
program kar;
var
  betu, csillag, szokoz: char;
begin
  betu := 'N';
  csillag := '*';
  szokoz := #32;
end.
```

### *3.2.1.4 Sorszámozott típusok*

Az egyszerű alaptípusokat osztályozhatjuk sorszámozott és nem sorszámozott típusokra. A sorszámozott típus értékkészletének elemei rendezetten, sorszámmal azonosítható módon helyezkednek el. Ennek köszönhetően egyértelműen meghatározható az előző (*preceding*) és a következő (*succeeding*) elem.

Az addig megismert sorszámozott típusok:

- az egész típusok, az ***int64*** és a ***qword*** típusok kivételével,
- a logikai típusok,
- a karakteres típusok.

A sorszámozott típusú adatokra három függvényt használhatunk:

- ***ord*** - a sorszámot szolgáltatja:

```
ord(false)   →   0
ord(true)    →   1
ord(13)      →   13
ord('A')     →   45
```

- ***succ*** – megadja a következő (azaz az eggyel nagyobb sorszámú) elemet:

```
succ(false)   →   true
succ(13)      →   14
succ('A')     →   'B'
```

- ***pred*** - az előző (azaz az eggyel kisebb sorszámú) elemet szolgáltatja:

```
pred(true)   →   false
pred(13)     →   12
pred('B')    →   'A'
```

A ***low*** függvénnyel az alsó, a ***high*** függvénnyel pedig a felső határát kapjuk meg a sorszámozott típus értékkészletének.

```
program p;
{$Mode FPC}
begin
  writeln(' az Integer típus legkisebb értéke : ',low(integer));
  writeln(' az Integer típus legnagyobb értéke: ',high(integer));
  writeln(' a Byte típus legnagyobb értéke    : ',high(byte));
  readln;
end.
```

*A program futásának eredménye:*

```
 az Integer típus legkisebb értéke : -32768
 az Integer típus legnagyobb értéke: 32767
 a Byte típus legnagyobb értéke    : 255
```

Sorszámozott típusú változók értékét az ***inc*** és a ***dec*** eljárások segítségével növelhetjük, illetve csökkenthetjük. Az eljárások első paramétere a változó, míg az opcionális második paraméter a lépéshosszt definiálja.

```
program incdec;

var
  x : integer = 12;
  c : char = 'O';
begin
  writeln(x:5);
  inc(x);       // növelés eggyel
  writeln(x:5);
  inc(x,10);    // növelés tízzel
  writeln(x:5);

  writeln(c:5);
  dec(c);       // csökkentés eggyel
  writeln(c:5);
  dec(c,2);     // csökkentés kettővel
  writeln(c:5);
  readln;
end.
```

*A program futásának eredménye:*

```
   12
   13
   23
    O
    N
    L
```

A Pascal nyelv két további sorszámozott típussal is rendelkezik, melyek értékkészletét magunknak kell megadnunk.

*A felsorolt típus*

A felsorolt típus elemeit kerek zárójelben, vesszővel elválasztva adjuk meg. Természetesen a felsorolásban minden azonosítónak egyedinek kell lennie. A felsorolás első elemének sorszáma 0.

```
var
   tantargy: (magyar, matematika, fizika, biologia);

      ord(magyar)        →        0
      ord(biologia)      →        3
      succ(magyar)       →        matematika
      pred(biologia)     →        fizika

      low(tantargy)      →        magyar
      high(tantargy)     →        biologia
```

Free Pascalban a *tantargy* változó tárolására kijelölt memóriaterület mérete a fordító módjától függ (***TP*** módban 1 bájt, míg ***FPC*** módban 4 bájt stb.). Azonban fordítási direktívával magunk is szabályozhatjuk a tárolási méretet (1, 2 vagy 4 bájt). Az alábbi formát alkalmazva; a *tantargy* változó mérete minden módban 1 bájt lesz:

```
{$PACKENUM 1}
var
   tantargy: (magyar, matematika, fizika, biologia);
```

A Free Pascal megengedi, hogy a felsorolásban az elemekhez (növekvő) értékeket rendeljünk, hasonlóan a C nyelv felsorolt típusához. A keletkező típus azonban nem tekinthető igazi sorszámozott típusnak, mivel az ilyen felsorolással nem használhatók a ***pred*** és a ***succ*** függvények:

```
program cenum;

{$PACKENUM 1}

type
    szinek = (fekete, kek, piros:=4, lila, sarga:=14, feher);

begin
  writeln(ord(low(szinek)));
  writeln(ord(high(szinek)));
  writeln(ord(piros));
  writeln(ord(lila));
  writeln(ord(sarga));
  writeln(sizeof(szinek));
  readln;
end.
```

*A program futásának eredménye:*

```
0
15
4
5
14
1
```

*A résztartomány típus*

Résztartomány típust más sorszámozott típus értékkészletének zárt intervallumaként definiáljuk. A típusban az alsó és a felső értékhatárt két ponttal választjuk el egymástól:

```
program felsorol;
var
  nagybetu: 'A'..'Z';
  index: 1..20;
begin
  nagybetu:= 'K';
  index:= 4;
end.
```

### 3.2.2 Karaktersorozat-típusok

A **string** típusú változókat karaktersorozatok tárolására használjuk. A Free Pascalban a hagyományos megoldások mellett az újabb platformok követelményeinek megfelelő típusokat is találunk. Az alábbi táblázatban összefoglaltuk a sztringtípusok legfontosabb jellemzőit:

| *Típusazonosító* | *Maximális hossz* | *Felhasználás* | *Nullavégű?* |
|---|---|---|---|
| ***ShortString*** | 255 | visszafelé kompatibilitás | nem |
| ***AnsiString*** | platformfüggő | ANSI szöveg tárolása | igen |
| ***String*** | 255, illetve platformfüggő | kompatibilis, illetve ANSI szöveg | nem, illetve igen |
| ***WideString*** | platformfüggő | Unicode kódolású szöveg tárolása | igen |
| ***PChar, PAnsiChar*** | platformfüggő | ANSI szöveg C-stílusú tárolása | igen |
| ***PWideChar*** | 1 gigabájt | Unicode szöveg C-stílusú tárolása | igen |

A fordító a **string** típust a *{$H}*, illetve a *{$LONGSTRINGS}* fordítási direktíva állapotától függően értelmezi. A direktíva bekapcsolt állapotában *{$H+}*/*{$LONGSTRINGS ON}* a **string** típus az ***AnsiString*** típust jelöli, míg kikapcsolva *{$H-}*/*{$LONGSTRINGS OFF}* a ***ShortString***-et.

Amennyiben a **string** típus mögött – szögletes zárójelben – a sztring hosszát is megadjuk, akkor a karaktersorozatot – a *{$LONGSTRINGS}* direktíva állapotától függetlenül –, rövid sztringként kezeli a fordító:

```
{$LONGSTRINGS ON}
var     rovid1  : ShortString;
        rovid2  : string[30];
        hosszu1 : string;
        hosszu2 : AnsiString;
```

A *{$LONGSTRINGS}* fordítási direktíva alapállapota a fordítási módtól függ. ***TP***, ***FPC*** és ***ObjFPC*** módokban *OFF*, míg ***Delphi*** módban *ON* állapotú.

A szövegkonstansokat, a karakterekhez hasonlóan, két aposztróf között adjuk meg, azonban használhatjuk a kódokkal történő megadást is, sőt a két formát vegyesen is alkalmazhatjuk:

```
'Pascal nyelv'   ''   ''''   'N'   #78#65#84#65   'Pascal'#32'nyelv'#0
```

A második konstans az ún. üres sztring, míg aposztrófot kettőzve helyezhetünk el a karaktersorozatban. Karaktersorozatokat a + művelettel is összekapcsolhatjuk.

```
program szovegek;
var
  varos: string;
  telefonszam: string[18];
begin
  varos := 'Budapest';
  telefonszam := '36-1-';
  telefonszam := telefonszam + '3751-564';
  writeln(varos);
  writeln(telefonszam);
  readln;
end.
```

*A program futásának eredménye:*

```
Budapest
36-1-3751-564
```

A sztring hosszának meghatározására a ***Length()*** függvényt használjuk:

```
hossz := Length('Lafenita');
```

***ShortString*** esetén a Free Pascal minden sztringhez egy külön bájtot rendel (0-ás indexszel), ahol megőrzi a karakterek aktuális darabszámát, vagyis a sztring hosszát.

Az ***AnsiString*** és a ***WideString*** típusú változók memóriaigénye mindig az aktuálisan tárolt szöveg méretétől függ. Ez a méret azonban a ***SetLength***() eljárással módosítható.

A **string** típusú változókban tárolt szöveget egy darabban és karakterenként egyaránt feldolgozhatjuk. A karakteres elérés esetén a sztringindexben (a sztring azonosítója mellett szögletes zárójelben) kell megadni a vizsgálandó karakter sorszámát. Az első karakter sorszáma 1.

```
nev    := 'Anna';     // Anna
nev[3]:= 'y';         // Anya
```

A szövegek platformfüggő felhasználását segítik a nullavégű (C-stílusú) sztringek támogatása, ahol a karaktersorozatot mutatóval azonosítjuk, és a szöveget egy nullaértékű bájt zárja.

```
program nullavegu;
var p : pchar;
begin
  p := 'Ez egy nullavégű szöveg';
  writeln (p);
  readln;
end.
```

A sztringek kezelését alprogramok gazdag gyűjteménye segíti a Free Pascalban. A *System* unitban a rövid és a hosszú sztringekkel egyaránt működő eljárásokat és függvényeket találunk. További hasznos rutinokat tartalmaz a *Sysutils* modul, míg a *Strings* unit a nullavégű karaktersorozatok használatát támogatja.

## 3.3 Típusnevek létrehozása, a type deklaráció

A Pascal programok deklarációs részében a **type** foglalt szó segítségével saját típusnevet is létrehozhatunk:

```
type
     új_típusnév = létező_típus;
```

vagy

```
type
     új_típusnév = type létező_típus;
```

Az első forma hatására az új típusnevet az eredeti típussal helyettesíti a fordító. Ha azonban a definíció mindkét oldalán szerepeltetjük a **type** foglalt szót, akkor a fordító a két típusnevet különbözőnek tekinti.

```
program typepr;
type
    egesz = integer;
    valos = real;
var
  a : egesz;
  b : valos;
```

```
begin
  a :=  12;
  b := -12.23;
end.
```

Megjegyezzük, hogy típusneveket elsősorban a résztartomány, a felsorolt és a strukturált típusok esetén hozunk létre, mivel az eljárások és a függvények ilyen típusú paramétereinek deklarációjában csak egyetlen típusnév szerepelhet.

## 3.4 Konstansok

A programnyelvek többsége lehetővé teszi, hogy a programban használt konstans értékekhez azonosítókat rendeljünk. Ez a megoldás jelentősen növeli a program forrásszövegének olvashatóságát, javíthatóságát.

### 3.4.1 Normál konstansok

A normál konstansok olyan azonosítóval ellátott értékek, amelyek nem változtathatók meg a program futása során. (Konstans neve nem szerepelhet az értékadó utasítás bal oldalán.)

A konstansokat a deklarációs rész **const** szekciójában hozzuk létre. A definícióban az egyenlőségjel bal oldalán a konstans neve áll, a jobb oldalán pedig a hozzárendelni kívánt érték (konstans kifejezés). A konstansnevet minden olyan értékhez definiálhatunk, amit a programban konstans értékként is megadhatunk.

```
const
  maxn  = $4000;                 // hexadecimális egész konstans (16384)
  minn  = &500;                  // oktális egész konstans (320)
  szokoz= chr(32);               // karakteres konstans
  eszam = 2.718281828459045235;// valós konstans
  udv   = 'Jó napot kívánok!'; // karaktersorozat-konstans
  maszk = %1001011011000011;   // bináris egész konstans (38595)
```

### 3.4.2 Típusos konstansok

A Free Pascal támogatja a típusos konstansok használatát, amelyek a memóriában tárolódnak és tartalmuk megváltoztatható. A típusos konstansok a Pascal nyelv kezdőértékkel ellátott statikus változói, melyek a program indításakor jönnek létre, és a programból való kilépésig léteznek.

A típusos konstansok szintén a deklarációs rész **const** szekciójában helyezkednek el, ahol a konstans kezdőértékét is meg kell adni:

```
const
     konstansnév : típus = kezdőérték;
```

Mivel Free Pascalban a változók is kaphatnak kezdőértéket, a típusos konstansok főprogram deklarációs részében a változókkal helyettesíthetők. Nem ez a helyzet azonban az alprogramokban definiált típusos konstansokkal, hisz ezek – a lokális változókkal ellentétben – a főprogram indításakor jönnek létre (8.4.1. alfejezet).

```
program konstans;
const
    fejlec: string = 'A feladat megoldása';
var
    max_ertek: integer = 1255;
begin
  writeln(fejlec);
  writeln(max_ertek);
  readln;
end.
```

*A program futásának eredménye:*

```
A feladat megoldása
1255
```

# 4. Alapvető I/O műveletek

A Pascal nyelvű programok alaphelyzetben az adatokat a billentyűzetről fogadják, a futási eredményeket pedig a képernyőn jelenítik meg. Az adatbevitelt (*input*) a ***read*** és a ***readln***, az eredmény megjelenítését (*output*) pedig a ***write*** és a ***writeln*** eljárások segítségével végezzük.

## 4.1 Írás a képernyőre - a write és a writeln eljárások

A szabványos kimenet (*standard output*) periféria a képernyő, amelyre két eljárással írhatunk:

```
write(paraméterek);
writeln(paraméterek);
```

*Megjegyzések:*

- a paramétereket vesszővel kell elválasztani,
- a paraméterek lehetnek:
  - változónevek,
  - konstansok,
  - aritmetikai, logikai és sztringkifejezések,
  - függvényhívás.

Mindkét eljárás megjeleníti a paraméterlistáján szereplő adatokat, azonban a ***writeln*** az adatok megjelenítését követően a szövegkurzort a következő sor elejére helyezi. Teszi ezt *DOS* és *Windows* alatt a #13#10 (*carriage return - linefeed*), *Unix* és *Linux* alatt pedig a #10 (*linefeed*) vezérlőkarakterek kiírásával. A paraméter nélküli ***writeln;*** hívás csak a szövegkurzort mozgatja.

Az eljárások hívásakor a paraméterek után formázási előírásokat is megadhatunk:

```
kifejezés : w
kifejezés : w : d
```

ahol *w* a megjelenítésnél használt mezőszélesség, míg valós értékek esetén a *d* a tizedes jegyek kívánt számát írja elő. Amennyiben az eredmény nem fér el a kijelölt mezőben, az eljárások figyelmen kívül hagyják az előírást. Érvényes mezőszélesség esetén az adatok a mező jobb széléhez igazodnak. A valós számok alaphelyzetben hatványkitevős alakban jelennek meg.

Az alábbi példaprogramok bemutatják a Pascal nyelv egyszerű típusú változóinak különféle módon való kiíratását:

- a mezőszélességet alkalmazva,
- több adat kiíratásával egyetlen ***writeln*** utasításban,
- a valós változók esetén a mezőszélesség és a tizedek számának megadásával,
- szöveges és logikai adatok megjelenítésével.

Írjunk programot, amely bemutatja az ***integer*** és a ***longint*** típusú változók tartalmának megjelenítését! (*iopr1*)

```
program iopr1;
var
  i,j,k: integer;
  h1,h2,h3: longint;
begin
  i:= 15;
  j:= 12500;
  k:= -45;
  h1:= 1235678;
  h2:= 34512389;
  h3:= -1234567889;
  writeln('12345678901234567890');
  writeln(i);  writeln(i:5);
  writeln(j);  writeln(j:8);
  writeln(k);  writeln(k:6);
  writeln(h1); writeln(h1:10);
  writeln(h2); writeln(h2:10);
  writeln(h3); writeln(h3:12);
  readln;
end.
```

*A program futásának eredménye:*

```
12345678901234567890
15
   15
12500
   12500
-45
   -45
1235678
   1235678
34512389
  34512389
-1234567889
 -1234567889
```

Készítsünk programot, amelyben a **real** típusú változók tartalmát formázva jelenítjük meg! (*iopr2*)

```
program iopr2;
var
  a,b,c,d,e: real;
begin
  a:= 2.56;
  b:= -0.005;
  c:= 123.893;
  d:= 1.65e-3;
  e:= 3.892345e4;
  writeln('12345678901234567890');
  writeln(a); writeln(a:10); writeln(a:10:2);
  writeln('--------------------');
  writeln(b); writeln(b:10); writeln(b:10:4);
  writeln('--------------------');
  writeln(c); writeln(c:10); writeln(c:10:3);
  writeln('--------------------');
  writeln(d); writeln(d:10); writeln(d:10:3);
  writeln('--------------------');
  writeln(e); writeln(e:10); writeln(e:10:2);
  readln;
end.
```

*A program futásának eredménye:*

```
12345678901234567890
 2.56000000000000E+000
 2.56E+000
      2.56
--------------------
-5.00000000000000E-003
-5.00E-003
   -0.0050
--------------------
 1.23893000000000E+002
 1.24E+002
   123.893
--------------------
 1.65000000000000E-003
 1.65E-003
    0.002
--------------------
 3.89234500000000E+004
 3.89E+004
  38923.45
```

Írjunk programot a ***boolean*** típusú változók tartalmának kiíratására! (*iopr3*)

```
program iopr3;
var
  b1, b2: boolean;
begin
  b1:= true;
  b2:= false;
  writeln('12345678901234567890');
  writeln(b1); writeln(b1:10);
  writeln('--------------------');

  writeln(b2); writeln(b2:10);
  writeln('--------------------');
  writeln(b1,b2);
  writeln('--------------------');
  writeln(b1:6,b2:6);
  writeln('--------------------');
  readln;
end.
```

*A program futásának eredménye:*

```
12345678901234567890
TRUE
      TRUE
--------------------
FALSE
     FALSE
--------------------
TRUEFALSE
--------------------
  TRUE FALSE
--------------------
```

Készítsünk programot, amelyben ***char*** és **string** típusú változók tartalmát jelenítjük meg! (*iopr4*)

```
program iopr4;
var
  c1, c2: char;
  s1, s2: string;

begin
  c1:= '*';
  c2:= 'A';
  s1:= 'szoveg';
  s2:= 'feladat';
  writeln('12345678901234567890');
  writeln(c1); writeln(c1:5);
  writeln('--------------------');
  writeln(c2); writeln(c2:2);
  writeln('--------------------');
```

```
  writeln(c1,c2); writeln(c1:2,c2:4);
  writeln('--------------------');
  writeln(s1);
  writeln('--------------------');
  writeln(s1:10);   writeln(s2:10);
  writeln('--------------------');
  writeln(s1:7,s2:8);
  writeln('--------------------');
  readln;
end.
```

*A program futásának eredménye:*

```
123456789012345678 90
*
    *
--------------------
A
 A
--------------------
*A
 *   A
--------------------
szoveg
--------------------
    szoveg
   feladat
--------------------
 szoveg feladat
--------------------
```

## 4.2 Olvasás billentyűzetről - a read és a readln eljárások

A ***read*** és a ***readln*** eljárások felfüggesztik a program futását, és várakoznak a billentyűzetről történő adatbevitelre. Az adatokat az *Enter* (↵) billentyű lenyomását követően dolgozzák fel:

***read***(*paraméterek*);

***readln***(*paraméterek*);

*Megjegyzések:*

- A paraméterek egész, valós, karakteres vagy sztringtípusú változók lehetnek.
- ***Boolean*** típusú változó értékét nem lehet billentyűzetről beolvasni!
- A paramétereket vesszővel kell elválasztani.
- Az adatok az *Enter* (↵) billentyű lenyomása után az adatbeviteli sorba kerülnek, amelynek a mérete platformfüggő.
- Több adat egyetlen sorban (*Enter* lenyomásáig) történő megadása esetén, az adatokat szóközzel kell tagolnunk.

A ***read*** eljárás az adatbeviteli sorból csak a paraméterei számára szükséges adatokat dolgozza fel, míg a fel nem használt rész egy következő ***read / readln*** eljárás számára elérhető marad.

Ezzel szemben a ***readln*** eljárás beolvassa a teljes adatsort, a bevitelt záró *Enter* (↵) billentyűnek megfelelő karakterekkel együtt, és ezt követően ad értéket a paraméterként megadott változóknak. Amennyiben minden paraméter értéket kapott, a beolvasott sor további része elvész. Ha azonban valamelyik paraméter értékadásához nem volt adat a sorban, újabb adatok bevitelére vár az eljárás.

Megjegyezzük, hogy az *Enter* (↵) billentyű lenyomásakor *DOS* és *Windows* alatt két karakter (*#13#10*), míg *Unix* és *Linux* alatt egyetlen jel (*#10*) kerül az adatbeviteli sorba.

Írjunk programot, amely egyetlen ***read*** eljárással beolvas három egész adatot! (*olvas1*)

A programnak kétféleképpen adjuk meg a három egész számot:
- egyetlen sorban, szóközzel tagolva,
- soronként egy-egy számot elhelyezve.

```
program olvas1;
var
  x, y, z: integer;
begin
  writeln('Egész változók olvasása read eljárással');
  read(x,y,z);
  writeln('Beolvasott adatok: ',x:4,y:4,z:4);
  readln; readln;
end.
```

*A program futásainak eredménye:*

```
Egész változók olvasása read eljárással
2 8 4↵
Beolvasott adatok:    2   8   4

Egész változók olvasása read eljárással
2↵
8↵
4↵
Beolvasott adatok:    2   8   4
```

Írjunk programot, amely három ***read*** eljárással olvas be 3 egész számot! (*olvas2*)

```
program olvas2;
var
  x, y, z: integer;
begin
  writeln('Egész változók olvasása read eljárással');
  read(x); read(y); read(z);
  writeln('Beolvasott adatok: ',x:4,y:4,z:4);
  readln; readln;
end.
```

*A program futásainak eredménye:*

```
Egész változók olvasása read eljárással
2 8 4↵
Beolvasott adatok:    2   8   4

Egész változók olvasása read eljárással
2↵
8↵
4↵
Beolvasott adatok:    2   8   4
```

Készítsünk programot, amely egyetlen ***readln*** eljárást használ három egész szám bevitelére! (*olvas3*)

```
program olvas3;
var
  x, y, z: integer;
begin
  writeln('Egész változók olvasása readln eljárással');
  readln(x,y,z);
  writeln('Beolvasott adatok: ',x:4,y:4,z:4);
  readln;
end.
```

*A program futásainak eredménye:*

```
Egész változók olvasása readln eljárással.
2 8 4↵
Beolvasott adatok:    2   8   4

Egész változók olvasása readln eljárással
2↵
8↵
4↵
Beolvasott adatok:    2   8   4
```

Írjunk programot, amely három ***readln*** eljárással olvassa be a három egész számot! (*olvas4*)

```
program olvas4;
var
  x, y, z: integer;
begin
  writeln('Egész változók olvasása readln eljárással');
  readln(x); readln(y); readln(z);
  writeln('Beolvasott adatok: ',x:4,y:4,z:4);
  readln;
end.
```

*A program futásainak eredménye:*

```
Egész változók olvasása readln eljárással
2 8 4↵
8↵
4↵
Beolvasott adatok:    2   8   4

Egész változók olvasása readln eljárással
2↵
8↵
4↵
Beolvasott adatok:    2   8   4
```

Az első futási eredményből jól látható, hogy a ***readln*** eljárás a teljes adatsort beolvassa, azonban ebből csak annyi adatot dolgoz fel, amennyi a paraméterlistáján szereplő változók számára szükséges.

Mivel a *readln(x);* az *x* változó értékadásához az adatsorból csak az első (2) adatot veszi ki, a megadott 8 és 4 számok elvesznek. Ezért a következő *readln(y);* eljárás az *y* változó feltöltésére új adatra vár.

Készítsünk programot, amely három ***read*** eljárással, négy ***char*** típusú változóba, beolvassa egy szöveg első négy karakterét! (*olvas5*)

```
program olvas5;
var
  c1,c2,c3,c4: char;
begin
  writeln('Karakterek olvasása read eljárással');
  read(c1,c2);
  read(c3); read(c4);
  writeln('Beolvasott adatok: ',c1,c2,c3,c4);
  readln; readln;
end.
```

*A program futásának eredménye:*

```
Karakterek olvasása read eljárással
alma↵
Beolvasott adatok: alma
```

Mint említettük, az adatbevitel során az *Enter* (↵) leütése is karaktert – *DOS* és *Windows* alatt két karaktert *#13* és *#10* – hoz létre, amit a ***read*** eljárással be is olvashatunk. Ez jól megfigyelhető, ha a fenti programot az alábbi formában futtatjuk:

```
Karakterek olvasása read eljárással
N↵
A↵
Beolvasott adatok: N
A
```

Amennyiben a beadott szöveg négy betűsnél hosszabb, a program végén álló első, paraméter nélküli ***readln;*** hívás a megmaradó szövegrészt „elnyeli":

```
Karakterek olvasása read eljárással
almafa↵
Beolvasott adatok: alma
```

Írjunk programot, amelyben két karaktert olvasunk be ***readln*** eljárással! (*olvas6*)

```
program olvas6;
var
  kod1, kod2: char;
begin
  writeln('Karakterek olvasása readln eljárással');
  write('KOD1: ');
  readln(kod1);
  write('KOD2: ');
  readln(kod2);
  writeln('Beolvasott kódok: ',kod1, kod2);
  readln;
end.
```

*A program futásainak eredménye:*

```
Karakterek olvasása readln eljárással
KOD1: *↵
KOD2: Q↵
Beolvasott kódok: *Q

Karakterek olvasása readln eljárással
KOD1: * / + -↵
KOD2: QKA↵
Beolvasott kódok: *Q
```

# 5. Operátorok és kifejezések

A Free Pascal kifejezések operandusok és műveleti jelek (operátorok) kombinációjából épülnek fel. Az operandusok lehetnek konstansok, változók, függvényhívások vagy újabb kifejezések. A kifejezés kiértékelésének célja, a kijelölt műveletek elvégzésével egyetlen érték meghatározása. A kifejezés (a kifejezés értékének) típusa függ az operandusok típusától és az elvégzett művelettől.

## 5.1 A műveleti jelek csoportosítása

A Pascal nyelvben a műveleteket az operandusok száma szerint két csoportba sorolhatjuk, az egy- és kétoperandusú műveletek csoportjába.

Amennyiben a művelet egyetlen operandus előtt szerepel, akkor az egyoperandusú (*unary*) operátor:

```
-y          not x
```

Ha az operandusok közrefogják a műveleti jelet, akkor kétoperandusú (*binary*) műveletről beszélünk:

```
y + 2       a < 12          n and b          'Free ' + 'Pascal'
```

A műveleteket azon kifejezés típusa alapján is csoportosíthatjuk, amelyen kifejtik a hatásukat. Ez alapján megkülönböztetünk:

- aritmetikai,
- relációs (összehasonlító),
- logikai,
- bit-,
- sztring-,
- halmaz-,
- egyéb (@, **is**, **as**) műveletek.

Néhány műveleti jel több csoportban is megtalálható. A fordítóprogram az operandusok száma és típusa alapján dönti el, hogy milyen műveletet kell végrehajtani.

A továbbiakban először összefoglaljuk az egyes csoportokhoz tartozó műveleti jeleket, majd pedig részletesebben megismerkedünk az egyszerű típusú adatokhoz kapcsolódó műveletekkel.

*A Pascal nyelv műveleti jeleinek csoportosítása:*

***aritmetikai műveletek***

| | |
|---|---|
| `-a` | előjelváltás, |
| `+a` | előjel megerősítés, |
| `a + b` | összeadás, |
| `a - b` | kivonás, |
| `a * b` | szorzás, |
| `a / b` | osztás, |
| `a div b` | egész osztás, |
| `a mod b` | maradék. |

***logikai műveletek***

| | |
|---|---|
| `not a` | tagadás, |
| `a and b` | ÉS, |
| `a or b` | VAGY, |
| `a xor b` | kizáró VAGY. |

***bitműveletek***

| | |
|---|---|
| `not a` | bitenkénti tagadás, |
| `a and b` | bitenkénti ÉS, |
| `a or b` | bitenkénti VAGY, |
| `a xor b` | bitenkénti kizáró VAGY, |
| `a shl b` | biteltolás balra, |
| `a shr b` | biteltolás jobbra. |

***sztringművelet***

| | |
|---|---|
| `a + b` | karaktersorozatok összekapcsolása. |

***halmazműveletek***

| | |
|---|---|
| `a + b` | unió, |
| `a * b` | metszet, |
| `a - b` | különbség, |
| `e in a` | eleme vizsgálat. |

***relációs műveletek***

| | |
|---|---|
| `a = b` | egyenlő, |
| `a <> b` | nem egyenlő, |
| `a < b` | kisebb, |
| `a <= b` | kisebb vagy egyenlő, |
| `a > b` | nagyobb, |
| `a >= b` | nagyobb vagy egyenlő. |

## 5.2 A műveletek elsőbbségi (precedencia) szabályai

A kifejezések általában egynél több műveletet is tartalmaznak, például:

```
3 * 5 + 8
```

Felvetődik a kérdés, hogy a kifejezés értéke *23* vagy *39* – ez attól függ, hogy melyik műveletet végezzük el előbb. Természetesen mi ismerjük a helyes választ (*23*), de honnan tudja a Pascal fordító, hogy először szorzunk, és csak aztán összegzünk?

A műveleti jelekhez tartozik egy ún. precedencia (elsőbbségi) tábla, amely egyértelműen meghatározza, hogy milyen sorrendben kell egy kifejezés műveleteit végrehajtani:

| *műveletei jelek* | *precedencia* |
|---|---|
| `@, not, -, +` | első (legmagasabb), |
| `*, /, div, mod, and, shl, shr, as` | második, |
| `+, -, or, xor` | harmadik, |
| `=, <>, <, >, <=, >=, in, is` | negyedik (legalacsonyabb). |

A táblázat sorai azonos elsőbbséggel rendelkező műveleteket tartalmaznak. Ha ilyen, azonos precedenciájú operátorból tartalmaz többet egy kifejezés, akkor a műveletek általában balról jobbra haladva hajtódnak végre. A Free Pascalban, bizonyos optimalizálási megfontolások következtében – például függvényhívás operandusok esetén – előfordulhat, hogy a végrehajtás sorrendje nem a balról jobbra lesz.

A műveletek kiértékelésének sorrendjét zárójelezéssel megváltoztathatjuk. A fentiekben szereplő kifejezés értéke 39 lesz, ha az összegzést kerek zárójelek közé tesszük:

```
3 * (5 + 8)
```

Zárójeleket akkor is érdemes használni, ha nem emlékszünk pontosan az elsőbbségi szabályokra. Az sem probléma, ha a zárójelezéssel csupán megerősítjük a kiértékelési szabályokat:

```
(3 * 5) + 8          (5 + 7) + 11
```

A zárójeleket egymásba skatulyázhatjuk, ekkor mindig a legbelső zárójeles kifejezés feldolgozásával indul a kiértékelés. Az alábbi kifejezés értéke *23*:

```
((1 + 2) * 3 + 4) + 10
```

## 5.3 Aritmetikai műveletek és kifejezések

Aritmetikai kifejezésben használjuk az aritmetikai műveleteket. A műveletek operandusai egész és valós típusúak egyaránt lehetnek.

| *operátor* | *művelet* | *a művelet egész számokkal* | *a művelet valós számokkal* |
|---|---|---|---|
| + | pozitív előjel | + 12 | + 12.23 |
| - | előjelváltás | - 23 | - 2.31E-1 |
| + | összeadás | 5 + 2 → 7 | 5.0 + 2.1 → 7.1 |
| - | kivonás | 5 - 2 → 3 | 5.3 - 3.0 → 2.3 |
| * | szorzás | 5 * 2 →10 | 5.1 * 3.2→ 16.32 |
| / | valós osztás | 5 / 2 → 2.5 | 4.83 / 2.3 → 2.1 |
| **div** | egész osztás | 5 **div** 2 → 2 | — |
| **mod** | maradékképzés | 5 **mod** 2 → 1 | — |

*Megjegyzések:*

- Az összeadás, kivonás és szorzás műveletek esetén, ha bármelyik operandus valós, akkor az eredmény is valós.
- Két egész szám hányadosának (/) az eredménye valós.
- A **div** és a **mod** műveletek csak egész számok esetén használhatók.

### 5.3.1 A precedencia-szabályok aritmetikai kifejezésekben

Mivel az aritmetikai műveleteket már a legegyszerűbb programokban is használjuk, külön összefoglaltuk Az elsőbbségi ún. precedencia szabályok pontosan meghatározzák a műveletek kiértékelésének a sorrendjét a kifejezésekben.

| *Operátorok* | *precedencia-szint* | *a művelet típusa* |
|---|---|---|
| -, + | első | egyoperandusú (előjel) |
| * , /, **div**, **mod** | második | multiplikatív |
| +, - | harmadik | additív |

Ha egy kifejezésben különböző elsőbbségű műveletek szerepelnek, akkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés értékelődik ki először. A táblázat soraiban szereplő műveletek között (általában) a balról-jobbra szabály jelöli ki a sorrendet.

Az $a+b*c-d*e$ és az $a+(b*c)-(d*e)$ kifejezések kiértékelési sorrendje megegyezik. Az alábbi értékadást és egész típusú változókat tételezve, kövessük nyomon az egyes lépéseket!

```
a:= 3; b:= 5; c:= 12; d:= 10; e:= 2;
```

1. `b * c -> 60`
2. `d * e -> 20`
3. `a + b * c = a + 60 -> 63`
4. `a + b * c - d * e = 63 - 20 -> 43`

A kiértékelés sorrendje a matematikából ismert zárójelek segítségével megváltoztatható. Megjegyezzük, hogy a Pascal nyelvben csak a kerek zárójel használható, bármilyen mélységű zárójelezést is alkalmazunk.

Az *(a + b) * (c - d) * e* kifejezés kiértékelésének lépései:

1. `(a + b) ->  8`
2. `(c - d) ->  2`
3. `8 * 2   -> 16`
4. `16 * 2  -> 32`

Ha azonos precedenciájú műveletek szerepelnek egy kifejezésben, akkor a balról-jobbra szabály lép életbe. Példaként tekintsük az alábbi kifejezést!

```
a + b * c / d * e;
```

A kifejezésben a két szorzás és az osztás azonos precedenciájú műveletek, ezért a kiértékelés sorrendje:

1. `b * c -> 60`
2. `60 / d -> 6.0`
3. `6.0 *e -> 12.0`
4. `a + 12.0 -> 15.0`

A kifejezés kiértékelésének lépéseit matematikai formára átírva jobban látható a műveletek sorrendje:

$$a+\frac{b\cdot c}{d}\cdot e$$

Ha azonban a feladat az alábbi képlet programozása, akkor ezt kétféleképpen is elvégezhetjük:

$$a+\frac{b\cdot c}{d\cdot e}$$

Zárójelbe tesszük a nevezőt, így mindenképpen a szorzattal osztunk:

```
a + b * c / (d * e);
```

```
1.  d * e    -> 20
2.  b * c    -> 60
3.  60 / 20  -> 3.0
4.  a + 3.0  -> 6.0
```

Az is jó megoldás, ha a nevezőben szereplő szorzat mindkét tagjával osztunk:

```
a + b * c / d / e;
```

```
1.  b * c    -> 60
2.  60 / 10  -> 6.0
3.  6.0 / 2  -> 3.0
4.  a + 3.0  -> 6.0
```

### 5.3.2 Matematikai függvények használata

Számítási feladatok programozása során, az aritmetikai műveleteken túlmenően matematikai függvényeket is használunk. A Free Pascal programokhoz automatikusan kapcsolódó ***System*** modul matematikai függvényeket is tartalmaz, melyek közül a legfontosabbakat az alábbi táblázatban foglaltuk össze:

| *Pascal hívás* | *matematikai forma* |
|---|---|
| *abs(x)* | $\|x\|$ |
| *arctan(x)* | arc tg x |
| *cos(x)* | cos x |
| *exp(x)* | $e^x$ |
| *ln(x)* | ln x |
| *sin(x)* | sin x |
| *sqr(x)* | $x^2$ |
| *sqrt(x)* | $\sqrt{x}$ |

Amennyiben az ***abs()*** és az ***sqr()*** függvények bemenő adata egész, akkor a visszatérési értékük is egész. Ha azonban a hívási paraméterük valós, akkor a függvények is valós eredményt szolgáltatnak. A többi függvény tetszőleges numerikus argumentum esetén valós eredménnyel tér vissza. Néhány további hasznos alprogram:

| *Pascal hívás* | *leírás* |
|---|---|
| *pi* | π értéke |
| *random(n)* | véletlen szám |
| *randomize;* | a véletlen szám generátor véletlenszerű indítása |

A Free Pascal ***math*** moduljában még több matematikai (statisztikai, számkonverziós, hatvány- és logaritmus-, hiperbolikus, trigonometriai valamint minimum- és maximum-meghatározó) függvények állnak a rendelkezésünkre.

Példaként tekintsük az egyismeretlenes, másodfokú egyenlet valós gyökeit meghatározó képlet Pascal nyelvre történő átírását! (Jelöljék az egyenlet együtthatóit *a, b* és *c* betűkkel!)

$$x_1 = \frac{-b+\sqrt{b^2-4\cdot a\cdot c}}{2\cdot a} \qquad x_2 = \frac{-b-\sqrt{b^2-4\cdot a\cdot c}}{2\cdot a}$$

```
program masodfoku;
var
  a, b, c, x1, x2 : real;
begin
  a:=1; b:=-35; c:=276;
  x1 := (-b+sqrt(sqr(b)-4*a*c)) / (2*a);
  x2 := (-b-sqrt(sqr(b)-4*a*c)) / (2*a);
  writeln('x1 = ', x1:8:3);
  writeln('x2 = ', x2:8:3);
  readln;
end.
```

### 5.3.3 Típus-átalakítások

A Pascal erősen típusos nyelv, így nem kell csodálkoznunk azon, hogy bizonyos típusú adatokat a fordítóprogram nem alakít át automatikusan más típusúvá. Ilyen átalakítások (szöveg → szám, valós → egész stb.) esetén „*Incompatible types*" fordítási hibajelzést kapunk. Természetesen, a kompatibilis típusok közötti típuskonverziók probléma nélkül végrehajthatók (például egész → valós, ***byte*** → ***integer***). Szükség esetén, az inkompatibilis típusok közötti átalakításokat többféle lehetőség közül választva végezhetjük el.

*Átalakítás a típusnevek segítségével (typecast)*

Pascalban a típus nevével megadott konverzió, csak korlátozottan használható. Az átalakítás általános formája:

```
típusnév(kifejezés)
```

Nézzünk néhány példát!

```
var
  c : char = 'N';
  p : pointer = nil;
  b : byte;
  v : boolean;
  a : cardinal;
```

```
begin
  b := byte(c);          // 78
  v := boolean(123);     // TRUE
  a := cardinal(p);      // 0
end.
```

*Átalakítás függvények felhasználásával*

A Free Pascal több könyvtári modulja is segíti a konverziós feladatok elvégzését. Helyszűke miatt, csak a leggyakrabban előforduló valós → egész, szám → szöveg, szöveg → szám átalakításokkal foglalkozunk.

Egy valós szám egész és törtrészét szolgáltatják a ***System*** modul a ***trunc()***, ***int()*** valamint a ***frac()*** függvényei. A két egész rész függvény közül a ***trunc()*** egész, míg az ***int()*** valós típusú eredményt ad. A valós-egész átalakítás alapfüggvénye a ***round()***, amely kerekítéssel állítja elő az egész típusú értéket.

A numerikus típusú adatok és a szövegben tárolt számok közötti átalakításhoz a ***System*** modul a ***str()*** és a ***val()*** eljárásokat biztosítja. Az első karaktersorozattá, míg a második számmá alakít. Sokkal kényelmesebben végezhetjük el a szükséges konverziókat a ***Sysutils*** modulban fellelhető függvényekkel:

| | |
|---|---|
| ***IntToStr()*** | egész → decimális ***string***, |
| ***IntToHex()*** | egész → hexadecimális **string**, |
| ***StrToInt()*** | dec. / hex. **string** → egész, |
| ***StrToIntDef()*** | dec. /hex. **string** → egész alapértékkel, melyet hiba esetén ad vissza, |
| ***FloatToStr()*** | valós → **string**, |
| ***FloatToText()*** | valós → nullavégű sztring, |
| ***FloatToStrF()*** | valós → **string**, formátumválasztással, |
| ***FormatFloat()*** | valós → string, formátummegadással, |
| ***StrToFloat()*** | string → valós, |
| ***TextToFloat()*** | nullavégű sztring → valós. |

Megjegyezzük, hogy az ***StrToFloat()*** és a ***TextToFloat()*** függvényeknek átadott karaktersorozatban tizedes pontot, illetve tizedes vesszőt egyaránt megadhatunk.

Az alábbi példaprogram jól szemlélteti az elmondottakat:

```
program strszam;
uses sysutils;
var
  v : real;
  e : integer;
  sv, se : string;
  hiba : word;
```

```
begin
  str(12.34, sv);          // valós szám szöveggé alakítása
  str(123, se);            // egész szám szöveggé alakítása
  val(sv,v,hiba);          // szövegből valós szám
  val(se,e,hiba);          // szövegből egész szám

  sv:=FloatToStr(12.34); // valós szám szöveggé alakítása
  se:=IntToStr(123);     // egész szám szöveggé alakítása
  v:=StrToFloat(sv);     // szövegből valós szám
  e:=StrToInt(se);       // szövegből egész szám
  readln;
end.
```

## 5.4 Összehasonlító és logikai műveletek

A Pascal nyelv feltételes utasításainak működését egy feltétel vezérli, melynek értéke igaz (**true**) vagy hamis (**false**) lehet. A feltételekben gyakran kell összehasonlítanunk bizonyos értékeket, hogy a program további működéséről döntsünk. Az összehasonlításokat az alábbi táblázatban szereplő összehasonlító (relációs) műveletekkel végezhetjük el. A relációs műveletek azonban csak egymással kompatíbilis típusú kifejezések (aritmetikai, szöveges stb.) összehasonlítására használhatók.

| *matematikai forma* | *Pascal kifejezés* | *jelentés* |
|---|---|---|
| a = b | a = b | *a* egyenlő *b*-vel, |
| a ≠ b | a <> b | *a* nem egyenlő *b*-vel, |
| a < b | a < b | *a* kisebb, mint *b*, |
| a ≤ b | a <= b | *a* kisebb vagy egyenlő, mint *b*, |
| a > b | a > b | *a* nagyobb, mint *b*, |
| a ≥ b | a >= b | *a* nagyobb vagy egyenlő, mint *b*. |

A táblázatban felsorolt ***boolean*** típusú Pascal kifejezések értéke **true**, amennyiben a vizsgált reláció igaz, illetve **false**, ha nem.

Példák számadatok összehasonlítására:

```
x:=5; y:=7; i:=25; j:=75;
x + y < 10    →   12 < 10   a reláció hamis
i + j = 100   →  100 = 100  a reláció igaz
```

Példák szövegek (karakterenkénti) összehasonlítására:

```
'alma' < 'almafa'     →  a reláció igaz
'Pascal' = 'Pascal'   →  a reláció igaz
```

További példaként írjunk fel egy olyan kifejezést, amely igaz (**true**) lesz, ha az *x* változó értéke nem negatív!

```
x >= 0
```

Hogyan lehet megvizsgálni, hogy az *x* értéke *-12* és *23* közé esik-e? Először a matematikából ismert módon írjuk fel a kifejezést:

```
-12 < x < 23
```

A kifejezést egy az egyben beírva egy Pascal programba, a fordításnál *„inkompatibilis típusok"* hibajelzést kapunk. Mi a teendő?

Ahhoz, hogy bonyolultabb feltételeket is meg tudjunk fogalmazni, a relációs műveletek mellett szükségünk van a logikai (*boolean*) operátorokra is. Pascalban az alábbi logikai műveletek használhatók a feltételek felírása során.

| *matematikai forma* | *Pascal kifejezés* | *jelentés* | *elsőbbségi szint* |
|---|---|---|---|
| ¬a | `not a` | tagadás (*NEM*) | *1.* |
| a ∧ b | `a and b` | logikai *ÉS*<br>(logikai szorzás - multiplikatív művelet) | *2.* |
| a ∨ b | `a or b` | logikai *VAGY*<br>(logikai összeadás - additív művelet) | *3.* |
| a ⊕ b | `a xor b` | logikai *kizáró VAGY*<br>(additív művelet) | *3.* |

A példánkban szereplő feltétel egyszerűen átírható Pascal kifejezéssé a logikai *ÉS* művelet alkalmazásával:

```
(-12 < x) and (x < 23)
```

(Zárójeleket kell használnunk, hisz az összehasonlító operátorok alacsonyabb precedenciával rendelkeznek a logikai operátoroknál.) Felhívjuk a figyelmet a Free Pascal által alkalmazott „rövidzár" kiértékelésre, ami azt jelenti, hogy a balról-jobbra haladó kiértékelés azonnal leáll, ha a kifejezés logikai értéke egyértelműen eldönthető. Az alábbi példában a *fv()* függvény csak akkor hívódik meg, ha a ***random(123)*** függvényhívás páros véletlen számmal tér vissza:

```
program rovidzar;

function fv() : boolean;
begin
  fv:=true;
end;

begin
  randomize;
  if (random(123) mod 2 = 0) and fv() then
    writeln('Páros véletlen szám');
  readln;
end.
```

Vannak esetek, amikor valamely feltétel felírása helyett egyszerűbb az ellentett feltételt megfogalmazni, és alkalmazni rá a logikai tagadás (*NEM*) operátorát (**not**). Az előző példában használt feltétel egyenértékű az alábbi kifejezéssel:

```
not((-12 >= x) or (x >= 23))
```

A logikai tagadás során minden relációt az ellentétes irányú relációra, az *ÉS* operátort pedig a *VAGY* operátorra (illetve fordítva) cseréljük.

A logikai (*boolean*) operátorok működését ún. igazságtáblával írjuk le:

| *a* | not *a* |
|---|---|
| false | true |
| true | false |

*logikai tagadás*

| *a* | *b* | *a* and *b* |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

*logikai ÉS művelet*

| *a* | *b* | *a* or *b* |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

*logika VAGY művelet*

| *a* | *b* | *a* xor *b* |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | false |

*logika kizáró VAGY művelet*

A műveletek lényegét szöveges formában is megfogalmazhatjuk:

| | |
|---|---|
| `not A` | A tagadás azt jelenti, hogy az *A* logikai értékének ellentettje lesz a kifejezés értéke. |
| `A and B` | A logikai **and** művelet csak akkor ad igaz eredményt, ha mindkét operandusa igaz. |
| `A or B` | A logikai **or** akkor ad igaz eredményt, ha legalább egyik operandusa igaz. |
| `A xor B` | A logikai **xor** akkor eredményez igaz értéket, ha pontosan az egyik operandusa igaz. |

Készítsünk programot, amelyben ***boolean*** típusú változókat különböző kifejezésekkel inicializálunk! Használjuk a relációs és a logikai műveleteket is! (*b_ertek*)

```
program b_ertek;
var
   x, y: real;
   b1, b2, b3, b4, b5, b6, b7, b8, b9, b10: boolean;
begin
  x:= 3; y:= 5;
  b1:= false;                                  // konstans érték
  b2:= true;                                   // konstans érték
  b3:= y > x;                                  // relációs kifejezés
  b4:= b2 and b3;                              // logikai kifejezés
  b5:= b1 or b2;                               // logikai kifejezés
  b6:= b2 or b3 and (x < y);                   // összetett feltétel
  b7:= (x > 2) and (x + y < 7);                // összetett feltétel
  b8:= (y - x < 3) and ( b1 or b2);            // összetett feltétel
  b9:= ((y < 5) or (x = 2)) and b2;            // összetett feltétel
  b10:= b5;                                    // boolean típusú változó
  writeln('b1: ',b1:6,' b2: ',b2:6,' b3: ',b3:6);
  writeln('b4: ',b4:6,' b5: ',b5:6,' b6: ',b6:6);
  writeln('b7: ',b7:6,' b8: ',b8:6,' b9: ',b9:6);
  writeln('b10:',b10:6);
  readln;
end.
```

*A program futásának eredménye:*

```
b1:  FALSE b2:   TRUE b3:   TRUE
b4:   TRUE b5:   TRUE b6:   TRUE
b7:  FALSE b8:   TRUE b9:  FALSE
b10:  TRUE
```

## 5.5 Bitműveletek

A magas szintű programozási nyelveken végzett munka során csak ritkán van szükség arra, hogy az adatokat a bájtnál kisebb egységek, a bitek segítségével kezeljünk. Ezért csak rövid áttekintést adunk a bitenként végzett műveletekről. A Free Pascal hat operátort bocsát rendelkezésünkre, melyekkel különböző bitenkénti logikai és biteltoló műveleteket hajthatunk végre az előjeles és az előjel nélküli egész adatokon.

### 5.5.1 Bitenkénti logikai műveletek

A műveletek első csoportja, a bitenkénti logikai műveletek, lehetővé teszik, hogy biteket teszteljünk, töröljünk vagy beállítsunk:

| *operátor* | *művelet* |
|---|---|
| **not** | 1-es komplemens, bitenkénti tagadás, |
| **and** | bitenkénti *ÉS,* |
| **or** | bitenkénti *VAGY,* |
| **xor** | bitenkénti kizáró *VAGY.* |

A bitenkénti logikai műveletek működésének leírását az alábbi táblázat tartalmazza, ahol a 0 és az 1 számjegyek a törölt, illetve a beállított bitállapotot jelölik.

| *a* | *b* | not *a* | *a* and *b* | *a* or *b* | *a* xor *b* |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |

A bitenkénti logikai műveletek biztosítják a számítógép hardverelemeinek programozását. A perifériák többségének alacsony szintű vezérlése bizonyos bitek beállítását, illetve törlését jelenti. A felhasznált műveleteket összefoglaló néven „maszkolásnak" nevezzük.

Minden egyes művelethez megfelelő bitmaszkot kell készítenünk, amellyel aztán logikai kapcsolatba hozva a megváltoztatni kívánt értéket, végbemegy a kívánt bitművelet. Mielőtt sorra vennénk a szokásos bitműveleteket, meg kell ismerkednünk az 1-, 2-, 4- és 8-bájtos adatelemek bitjeinek sorszámozásával. A bitek sorszámozása mindig a legkisebb helyértékű bittől indulva 0-val kezdődik, és balra haladva növekszik. Mivel a példáinkban 16-bites ***smallint*** típusú adatelemeket használunk, nézzük meg ezen adatok felépítését!

| 15. | 14. | 13. | 12. | 11. | 10. | 9. | 8. | 7. | 6. | 5. | 4. | 3. | 2. | 1. | 0. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

A tárolt szám értéke hexadecimális számrendszerben *$07D6*, illetve decimális alakban *2006*.

Az alábbi példákban a ***smallint*** típusú *2525* szám *7.* és *12.* bitjeit kezeljük:

```
a := 2525;  // $09DD, %0000100111011101
```

| *Művelet* | *maszk* | *Pascal utasítás* |
|---|---|---|
| bitek 1-be állítása | %0001000010000000, $1080 | e:=a **or** $1080; |
| bitek törlése | %1110111101111111, $EF7F | e:=a **and** $EF7F; |
| bitek negálása | %0001000010000000, $1080 | e:=a **xor** $1080;<br>e:=e **xor** $1080; |
| az összes bit negálása (kétféleképpen) | —<br>%1111111111111111, $FFFF | e := **not** a;<br>e:=a **xor** $FFFF; |

A kijelölt műveleteket az alábbi program segítségével végeztük el. Az eredményeket bináris, hexadecimális és decimális formában egyaránt megjelenítettük.

```
program maszkolas;
var
  a, e : smallint;

begin
  a := 2525;
  writeln('a =          %', binstr(a,16),'  $',hexstr(a,4),'  ',a:5);
  e := a or $1080;
  writeln('beállítás    %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  e := a and $EF7F;
  writeln('törlés       %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  e := a xor $1080;
  writeln('negálás 1x   %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  e := e xor $1080;
  writeln('negálás 2x   %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  e := not a;
  writeln('mind negálás %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  e:=a xor $FFFF;
  writeln('mind negálás %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  readln;
end.
```

*A program futásának eredménye:*

```
a =          %0000100111011101  $09DD   2525
beállítás    %0001100111011101  $19DD   6621
törlés       %0000100101011101  $095D   2397
negálás 1x   %0001100101011101  $195D   6493
negálás 2x   %0000100111011101  $09DD   2525
mind negálás %1111011000100010  $F622  -2526
mind negálás %1111011000100010  $F622  -2526
```

Felhívjuk a figyelmet a kizáró vagy operátor (**xor**) érdekes tulajdonságára. Ha ugyanazzal a maszkkal kétszer végezzük el a műveletet, visszakapjuk az eredeti értéket, esetünkben a *2525*-öt.

### 5.5.2 Biteltoló műveletek

A bitműveletek másik csoportjába, a biteltoló (*shift*) operátorok tartoznak. Az eltolás balra (**shl**) és jobbra (**shr**) egyaránt elvégezhető. Az eltolás során a bal oldali operandus bitjei annyiszor lépnek balra (jobbra), amennyi a jobb oldali operandus értéke. A felszabaduló bitpozíciókba 0-ás bitek kerülnek, míg a kilépő bitek elvesznek. A példában ugyancsak a ***smallint*** típusú *a* változót használjuk:

| *Művelet* | *Pascal utasítás* |
|---|---|
| 2 bittel való eltolás balra, | `a := 2525;  e := a shl 2;` |
| 3 bittel való eltolás jobbra (nem negatív), | `a := 2525;  e := a shr 3;` |
| 3 bittel való eltolás jobbra (negatív). | `a := -2525; e := a shr 3;` |

A táblázatban szereplő műveleteket elvégző Pascal program:

```
{$R-}
program biteltolas;
var
  a, e : smallint;
begin
  a := 2525; e := a shl 2;
  writeln('a =          %', binstr(a,16),'  $',hexstr(a,4),'  ',a:5);
  writeln('a shl 2      %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  writeln;
  a := 2525; e := a shr 3;
  writeln('a =          %', binstr(a,16),'  $',hexstr(a,4),'  ',a:5);
  writeln('a shr 3      %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  writeln;
  a := -2525; e := a shr 3;
  writeln('a =          %', binstr(a,16),'  $',hexstr(a,4),'  ',a:5);
  writeln('a shr 3      %', binstr(e,16),'  $',hexstr(e,4),'  ',e:5);
  readln;
end.
```

*A program futásának eredménye:*

```
a =          %0000100111011101  $09DD   2525
a shl 2      %0010011101110100  $2774  10100

a =          %0000100111011101  $09DD   2525
a shr 3      %0000000100111011  $013B    315

a =          %1111011000100011  $F623  -2525
a shr 3      %1111111011000100  $FEC4   -316
```

Az eredményeket megvizsgálva láthatjuk, hogy a 2 bittel való balra eltolás során az *a* változó értéke négyszeresére ($2^2$) nőtt, míg három lépéssel jobbra eltolva, *a* értéke nyolcad ($2^3$) részére csökkent. Általánosan is megfogalmazható, hogy valamely egész szám bitjeinek *n* lépéssel történő balra tolása a szám ($2^n$) értékkel való megszorzását eredményezi. Az *m* bittel való jobbra eltolás pedig ($2^m$) értékkel elvégzett egész osztásnak felel meg. (Megjegyezzük, hogy a jobbra való eltolás előjelhelyesen megy végbe.)

Írjunk programot, amely a beolvasott 16-bites egész számot két bájtra bontja! A feladat aritmetikai és bitenkénti műveletek felhasználásával egyaránt megoldható. Nézzünk egy lehetséges megoldást, amelyben bitműveleteket használtunk! (*darabol*)

```
program darabol;
uses sysutils;

var
  num : word;
  lo, hi : byte;
```

```
begin
  // A szám beolvasása
  write('Kérek egy egész számot [0,65535] : ');
  readln(num);
  // Az alsó bájt meghatározása maszkolással
  lo := num and $00FF;
  writeln('A szám alsó bájtja:  $', hexstr(lo,2));
  // Az felső bájt meghatározása biteltolással
  hi := num shr 8;
  writeln('A szám felső bájtja: $', hexstr(hi,2));
  readln;
end.
```

*A program futásainak eredménye:*

```
Kérek egy egész számot [0,65535] : %1010010111110000
A szám alsó bájtja:  $F0
A szám felső bájtja: $A5

Kérek egy egész számot [0,65535] : $1223
A szám alsó bájtja:  $23
A szám felső bájtja: $12
```

## 5.6 Sztringműveletek

A Pascal nyelv operátorai közül csak a + és a relációs műveletek használhatók karaktersorozat (***ShortString***, ***AnsiString*** és ***WideString***) típusú operandusokkal. A + művelet összekapcsolja az operandusaiban szereplő karaktersorozatokat egyetlen sztringgé. A nullavégű (***PChar***) sztringek esetén egyetlen művelet sem alkalmazható.

Az alábbi példában a műveletekkel támogatott típusú karaktersorozatokat kapcsolunk össze:

```
var
  rs : shortstring = 'Pascal';
  as : ansistring  = 'Pascal';
  ws : widestring  = 'Pascal';
begin
  rs := 'Free ' + rs;
  as := 'Free ' + as;
  ws := 'Free ' + ws;
  as := rs + ':' + as;
  ws := ws + '|' + rs +  '|' + as;
  writeln(rs > as);    // FALSE
  writeln(as = as);    // TRUE
  writeln(ws > as);    // TRUE
  readln;
end.
```

Az operátorok mellett a Free Pascal ***System***, ***SysUtils***, ***StrUtils*** és ***Strings*** moduljainak alprogramjai is segítik a szövegkezelési feladatok programozását.

Az alábbi táblázatban összefoglaltuk a ***System*** modulban definiált sztringkezelő függvényeket és eljárásokat, amelyek rövid, hosszú és széles sztringek esetén egyaránt felhasználók:

| | |
|---|---|
| ***BinStr()***, ***HexStr()***, ***OctStr()*** | egész szám sztringgé alakítása különböző számrendszerben, |
| ***Chr()*** | ANSI karakter előállítása a kódjából, |
| ***Concat()*** | sztringek összefűzése, |
| ***Copy()*** | sztringrészlet másolása, |
| ***Delete()*** | sztringrész törlése, |
| ***Insert()*** | sztring beszúrása, |
| ***Length()*** | a sztring hosszának lekérdezése, |
| ***Lowercase()*** | a sztring kisbetűssé alakítása, |
| ***Pos()*** | sztringben sztringrész keresése, |
| ***SetLength()*** | sztring hosszának módosítása, |
| ***SetString();*** | sztring tartalmának és hosszának beállítása, |
| ***Str()*** | szám sztringgé alakítása, |
| ***StringOfChar()*** | karakterből adott hosszú sztring előállítása, |
| ***Upcase()*** | a sztring nagybetűssé alakítása, |
| ***Val()*** | sztring számmá alakítása. |

## 5.7 Példaprogramok egyszerű számítások elvégzésére

A Pascal alapműveleteinek könnyebb elsajátítását célozzák a fejezetet záró példaprogramok.

Készítsünk programot, amely beolvassa a derékszögű háromszög átfogóját és a rajta lévő egyik szöget. Számítsuk ki a két befogót! (*derekszh*)

```
program derekszh;
var
   a, b, c, alfa: real;

begin
  writeln('Derékszögű háromszög adatai');
  write('c átfogó : '); readln(c);
  write('alfa szög: '); readln(alfa);
  a:= c*sin(alfa*pi/180);
  b:= c*cos(alfa*pi/180);
  writeln('a oldal: ',a:5:2);
  writeln('b oldal: ',b:5:2);
  readln;
end.
```

*A program futásának eredménye:*

```
Derékszögű háromszög adatai
c átfogó : 5
alfa szög: 30
a oldal:  2.50
b oldal:  4.33
```

Írjunk programot, amely beolvassa egy általános háromszög két oldalát és a közbezárt szöget! Határozzuk meg a harmadik oldal hosszát koszinusz-tétellel! (*haromsz*)

```
program haromsz;
var
   a,b,c,gamma: real;
begin
  writeln('Általános háromszög adatai');
  write('a oldal: '); readln(a);
  write('b oldal: '); readln(b);
  write('közbezárt szög: '); readln(gamma);
  c := sqrt(sqr(a)+sqr(b)-2*a*b*cos(gamma*pi/180));
  writeln('c oldal: ',c:5:2);
  readln;
end.
```

*A program futásának eredménye:*

```
Általános háromszög adatai
a oldal: 3
b oldal: 4
közbezárt szög: 30
c oldal:  2.05
```

Készítsünk programot, amely beolvassa egy kör sugarát, és kiszámítja a kör területét és a kerületét! (*kortk*)

```
program kortk;
var
   sugar:real;
   terulet, kerulet: real;
begin
  write('A kör sugara: '); readln(sugar);
  terulet:= sqr(sugar)*pi;
  kerulet:=2*sugar*pi;
  writeln('A kör területe: ', terulet:10:4);
  writeln('A kör kerülete: ', kerulet:10:4);
  readln;
end.
```

*A program futásának eredménye:*

```
A kör sugara: 1
A kör területe:      3.1416
A kör kerülete:      6.2832
```

Írjunk programot, amely beolvassa a gömb sugarát, majd pedig kiszámítja a gömb felszínét és térfogatát! (*gombft*)

```
program gombft;
var
   sugar:real;
   felszin, terfogat: real;
begin
  write('A gömb sugara: '); readln(sugar);
  felszin:= 4*sqr(sugar)*pi;
  terfogat:=4*sqr(sugar)*sugar*pi/3;
  writeln('A gömb felszíne : ', felszin:10:4);
  writeln('A gömb térfogata: ', terfogat:10:4);
  readln;
end.
```

*A program futásának eredménye:*

```
A gömb sugara: 1
A gömb felszíne :     12.5664
A gömb térfogata:      4.1888
```

Írjunk programot, amely egész számként beolvas egy mm-ben megadott távolságot! Alakítsuk át a távolságot méter, centiméter és milliméter egységekké! (*mmvaltas*)

```
program mmvaltas;
var
   tav, mm, m, cm: longint;
begin
  writeln('Kérek egy egész számot!');
  write('távolsag [mm]: '); readln(tav);
  m := tav div 1000;
  mm:= tav mod 10;
  cm:= tav mod 1000 div 10;
  writeln(m, ' m ',cm, ' cm ',mm,' mm');
  readln;
end.
```

*A program futásának eredménye:*

```
Kérek egy egész számot!
távolsag [mm]: 12534
12 m 53 cm 4 mm
```

Írjunk programot, amely beolvassa két síkbeli pont koordinátáit, majd pedig meghatározza a két pont távolságát! (*tav*)

```
program tav;
var
   x1, y1, x2, y2:integer;
   tavolsag: real;
```

```
begin
  write('1. pont x koordinátája: '); readln(x1);
  write('1. pont y koordinátája: '); readln(y1);
  writeln;
  write('2. pont x koordinátája: '); readln(x2);
  write('2. pont y koordinátája: '); readln(y2);
  tavolsag:= sqrt(sqr(abs(x1-x2))+sqr(abs(y1-y2)));
  writeln('Távolság: ',tavolsag:10:6);
  readln;
end.
```

*A program futásának eredménye:*

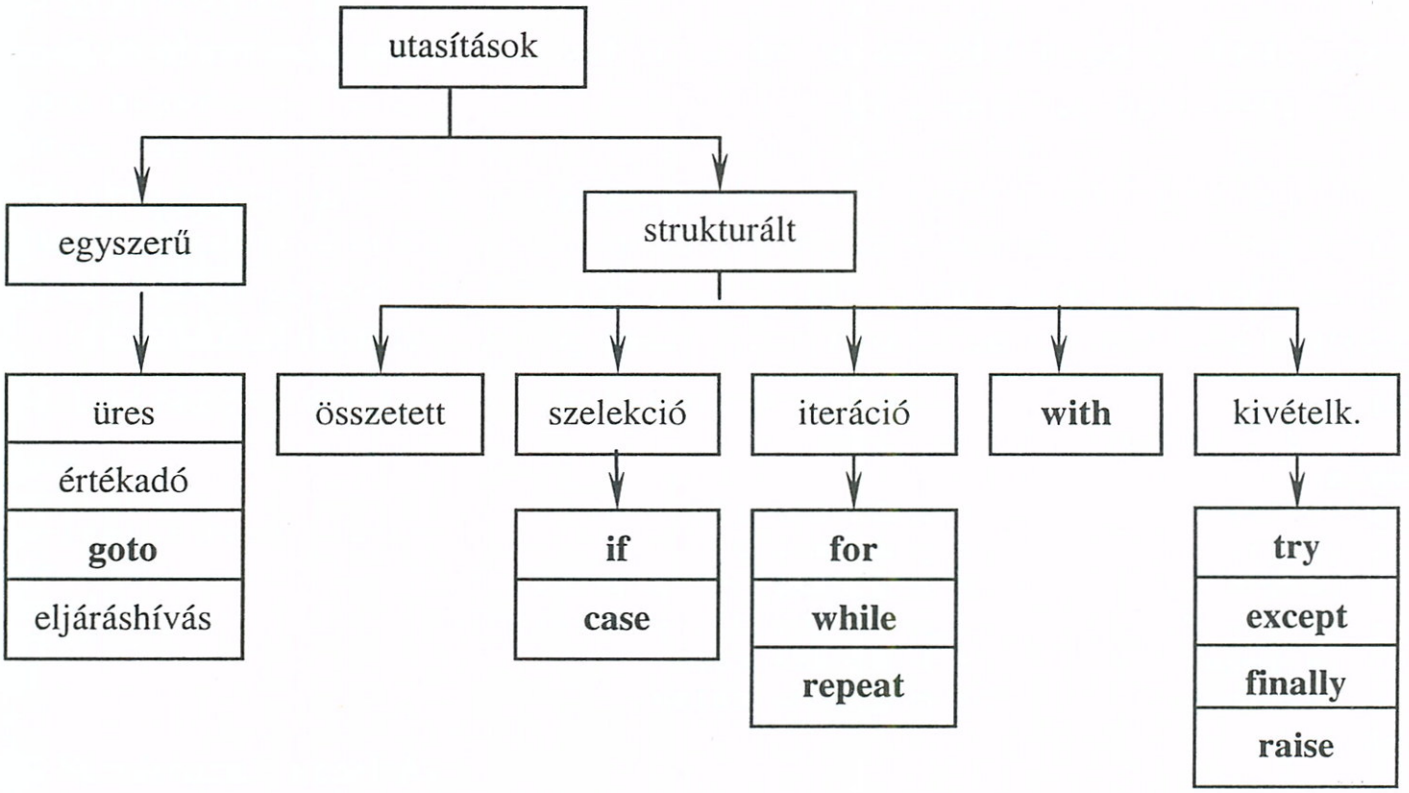
```
1. pont x koordinátája: 2
1. pont y koordinátája: 2

2. pont x koordinátája: 1
2. pont y koordinátája: 1
Távolság:   1.414214
```

# 6. A Pascal nyelv utasításai

A Pascal nyelven írt program végrehajtható része elvégzendő tevékenységek (utasítások) sorozatából áll. A programokban az egyszerű utasítássorozaton (szekvencián) túlmenően bonyolultabb programszerkezetek is megvalósíthatók.

A Pascal nyelv utasításait többféleképpen csoportosíthatjuk. A szelekciós utasítások az egyszerű, illetve többszörös elágazást lehetővé tevő **if** és **case** utasításokat jelölik. Az iterációs utasítások csoportja a Pascal nyelv ciklusutasításait (**while**, **for**, **repeat-until**) tartalmazza. A kivételkezeléssel kapcsolatos (**try**, **except**, **finally** és **raise**) utasítások csak ***ObjFPC*** és ***Delphi*** fordítási módban használhatók.




A táblázatban nem szerepel az **asm** utasítás, melynek segítségével egy sorban, vagy az **asm...end** blokkban assembly utasításokat szúrhatunk a Pascal kódba. (Könyvünkben eltekintünk az alacsony szintű programozás nyelvének ismertetésétől.) A **with** és kivételkezelés utasításaival csak a későbbi fejezetekben foglalkozunk.

## 6.1 Egyszerű utasítások

Az egyszerű utasítások segítségével egyetlen lépésben elvégezhető tevékenységeket írhatunk elő. Az eddigi példaprogramjaink csak ilyen utasításokat tartalmaznak. Komolyabb algoritmus programozásához azonban elengedhetetlen az összetettebb programstruktúrák alkalmazása.

### 6.1.1 Az üres utasítás

```
;
```

Az üres utasítás csupán egy jelzés arra, hogy a program semmilyen tevékenységet sem végez. Az üres utasítást akkor használjuk, ha a programban csak a Pascal nyelv előírásainak megfelelően kell egy utasítást elhelyeznünk. Az üres utasítás nem a pontosvessző, hanem a „semmi" az utasítást záró pontosvessző előtt:

```
begin
  writeln('Most üres utasítások következnek:');
  ;;     // két üres utasítás
  ;      // egy üres utasítás
end.
```

### 6.1.2 Az értékadó utasítás

A leggyakrabban használt Pascal utasítás az értékadás, melynek általános formája:

```
változónév := kifejezés;
```

Az utasítás végrehajtása során, a bal oldalon megadott változó felveszi a jobb oldali kifejezés értékét. Az utasítás csak akkor fordítható le, ha fennáll az értékadási kompatibilitás a két oldal típusai között. (Felhívjuk a figyelmet arra, hogy az értékadó jel (:=) karakterei közé nem szabad szóközt írni!) Néhány példát az utasítás használatára!

```
var
   x, y : integer; b : boolean;
begin
  x := 12;              // x értéke 12
  y := 23;              // y értéke 23
  x := x + y;           // x értéke 35, y értéke 23
  y := y - 7;           // x értéke 35, y értéke 16
  b := true;            // b értéke true
  b := not b;           // b értéke false
end.
```

### 6.1.3 A goto utasítás

A hibátlan, áttekinthető programszerkezet „ellenségének" tekintett **goto** utasítás feltétel nélküli vezérlésátadást (ugrást) valósít meg, a címkével megjelölt programsorra.

```
goto címke;

címke:
```

A *címke* Pascal azonosító, vagy decimális számjegyek sorozata, amelyet előzőleg a deklarációs rész **label** szekciójában tettük ismertté a fordítóprogram számára,

```
label
  kilep, 1223;
```

majd pedig – kettősponttal lezárva – elhelyeztük a deklarációhoz tartozó programrészben:

```
begin
   1223:   writeln ;
   kilep:
           readln;
end.
```

Free Pascalban; strukturált utasítások alkalmazásával megvalósíthatjuk a **goto** utasítások nélküli programozást. Sőt a fordító beállításaival (***Options/Compiler...***) le is tilthatjuk a **label** és **goto** foglalt szavak használatát.

### 6.1.4 Eljáráshívás

Egy adott tevékenység – paraméterezett – elvégzésére szolgáló, névvel ellátott programegységeket eljárásoknak nevezzük. Az eljárások csak akkor végzik el a beprogramozott algoritmust, ha hívjuk őket. Az eljáráshívás folyamán a hívott programrész lefutása után a vezérlés visszakerül a hívás utáni utasításra. A hívás lehetséges formái:

```
eljárásnév(hívási paraméterek);
eljárásnév;
eljárásnév();
```

Példaként nézzünk néhány jól ismert Pascal eljáráshívást!

```
begin
  writeln();
  writeln('Free Pascal');
  writeln;
  readln;
  readln();
end.
```

## 6. 2 Strukturált utasítások

A strukturált utasítások más, egyszerű és strukturált utasításokból épülnek fel. A utasítások által meghatározott tevékenységek

- egymás után (az összetett, a **with** és a **try...except/finally** utasításokban),
- feltételtől függően (az **if** és a **case** utasításokban),
- ciklikusan ismétlődve (a **for**, a **while** és a **repeat** utasításokban)

hajtódnak végre.

### 6.2.1 Összetett utasítás (blokk)

A **begin** és az **end** foglalt szavak között megadott utasítássorozatot összetett utasításnak (utasításblokknak) nevezzük.

```
begin
  utasítás1;
  utasítás2;
    ...
end;
```

Az összetett utasítást akkor használjuk, ha több utasítás végrehajtására van szükség, azonban Pascal nyelv csak egyetlen utasítás megadását teszi lehetővé. Az összetett utasításban szereplő **begin** és **end** foglalt szavakat utasítás-zárójelnek is szokás hívni. A blokkon belüli utolsó utasítást nem szükséges pontosvesszővel lezárni:

```
var
  a,b,c : integer;
begin
  // ...
  begin
     c := a;
     a := b;
     b := c
  end;
end.
```

### 6.2.2 Feltételes utasítások - szelekció

Az algoritmusok programozása során gyakran előfordul, hogy a végrehajtandó tevékenység adott pontján – feltételtől függően – két vagy több irányban kell a programot elágaztatnunk. Ennek megvalósítására az **if** és a **case** utasításokat használjuk.

#### *6.2.2.1 Az if utasítás*

Az **if-then-else** utasítás kétirányú feltételes elágazást tesz lehetővé. Az utasítás általános formája:

```
if feltétel then utasítás1 else utasítás2;
```

Az **if** után álló *feltétel* egy logikai kifejezés, melynek értékétől függ, hogy melyik utasítással fut a program tovább. Ha a *feltétel* értéke igaz (**true**), akkor a **then** utáni *utasítás1*, ellenkező esetben (**false**) pedig az **else** ágban álló *utasítás2* hajtódik végre.

Az **if-then-else** struktúrában az **else** előtt nem állhat pontosvessző, mert akkor a fordító az adott utasítást az **if** rövidített változatának tekinti. Mivel az **else** foglalt szóval önálló utasítás nem kezdődhet, a fordítás hibajelzéssel megszakad.

Az **if** utasítás rövid alakja:

```
if feltétel then utasítás;
```

Amennyiben a feltétel nem teljesül (**false**), akkor a vezérlés egyszerűen átadódik az **if** után álló utasításra.

Az **if** szerkezet mindkét ágában pontosan egy utasítás szerepelhet. Ha például, a **then** után semmilyen tevékenységet sem kívánunk megadni, akkor az üres utasítást kell használnunk.

```
if feltétel then else utasítás;
```

Természetesen a fenti vizsgálat egyszerűbben is elvégezhető a feltétel tagadásával:

```
if not feltétel then utasítás;
```

Amennyiben a feltételtől függően egynél több utasítást kell végrehajtanunk, akkor az utasításokat **begin-end** blokkba kell helyeznünk:

```
if feltétel then
    begin
      // utasítások
    end
 else
    begin
      // utasítások
    end;
```

```
if feltétel then
    begin
      // utasítások
    end;
```

Többirányú elágazást az **if** utasítások egymásba építésével valósíthatunk meg:

```
if feltétel1 then
     utasítás1
else if feltétel2 then
         utasítás2
     else
         utasítás3;
```

Előfordulhat, hogy az **if** teljes és rövid formáját építjük egybe. Ekkor kérdésessé válik, hogy az **else** melyik **if** utasításhoz tartozik. Az alábbi utasítás az *a* változó negatív értékei esetén semmit sem ír ki, *0* esetén megjeleníti a „*Negatív*" szöveget, míg pozitív értékekkel helyesen működik:

```
if a>=0 then
   if a>0 then
     writeln('Pozitív')
else
  writeln('Negatív');
```

A hibás működés oka, hogy az **else** – a szándékaink ellenére – a belső (legközelebbi, előtte álló) **if**-hez tartozik. A helyesen működő utasításhoz, amelyik csak a negatív és

a pozitív számokat különbözteti meg, két úton is eljuthatunk: használhatjuk az üres utasítást, vagy alkalmazhatjuk az utasításblokkot.

```
if a>=0 then
   if a>0 then
     writeln('Pozitív')
   else
else
  writeln('Negatív');
```

```
if a>=0 then
  begin
    if a>0 then
       writeln('Pozitív')
  end
else
  writeln('Negatív');
```

Nézzünk néhány példát az **if** utasítás alkalmazására!

Írjunk programot, amely beolvas egy egész számot, és írja ki, hogy a szám páros, vagy páratlan! (*parossag*)

```
program parossag;
var
  szam: integer;
begin
  write('Kérek egy egész számot: '); readln(szam);
  if szam mod 2 = 0 then
      writeln(szam,' páros')
  else
      writeln(szam,' páratlan');
  readln;
end.
```

*A program futásainak eredményei:*

```
Kérek egy egész számot: 12
12 páros

Kérek egy egész számot: 13
13 páratlan
```

Készítsünk programot, amely beolvas három valós számot, és kiválasztja, majd megjeleníti közülük a legnagyobbat! (*max3*)

```
program max3;
var
  szam1, szam2, szam3, legnagyobb: real;
begin
  writeln('Kérek három valós számot!');
  write('1. adat: '); readln(szam1);
  write('2. adat: '); readln(szam2);
  write('3. adat: '); readln(szam3);
  legnagyobb := szam1;
  if szam2 > legnagyobb then legnagyobb := szam2;
  if szam3 > legnagyobb then legnagyobb := szam3;
```

```
  writeln('A legnagyobb: ',legnagyobb:6:1);
  readln;
end.
```

*A program futásainak eredménye:*

```
Kérek három valós számot!
1. adat: 4
2. adat: 1
3. adat: 3
A legnagyobb:     4.0

Kérek három valós számot!
1. adat: 5
2. adat: 12
3. adat: 3
A legnagyobb:    12.0
```

Írjunk programot, amely beolvas három egész számot, és növekvő sorrendben megjeleníti a számokat! (*sorrend3*)

```
program sorrend3;
var
  a, b, c : integer;
begin
  writeln('Kérek három egész számot!');
  write('1. szám: '); readln(a);
  write('2. szám: '); readln(b);
  write('3. szám: '); readln(c);
  if a<=b then
     if  b<=c then
         writeln(a:5, b:5, c:5)
     else
        if a<=c then
           writeln(a:5, c:5, b:5)
        else
           writeln(c:5, a:5, b:5)
  else
     if a<=c then
           writeln(b:5, a:5, c:5)
     else
        if b<=c then
           writeln(b:5, c:5, a:5)
        else
           writeln(c:5, b:5, a:5);
  readln;
end.
```

Természetesen, adatok sorba rendezésére a későbbiekben sokkal hatékonyabb algoritmusok közül is választhatunk.

*A program futásainak eredménye:*

```
Kérek három egész számot!
1. szám: 23
2. szám: 7
```

```
3. szám: 12
    7   12   23

Kérek három egész számot!
1. szám: -3
2. szám: 5
3. szám: 0
   -3    0    5
```

Készítsünk programot, amely beolvassa egy háromszög három oldalának hosszát, megvizsgálja, hogy az oldalhosszak nagyobbak-e nullánál, valamint, hogy a háromszög megszerkeszthető-e! Számítsuk ki a háromszög kerületét, valamint a területét a *Heron*-képlettel! (*haromszog*)

A háromszög akkor szerkeszthető meg, ha bármely két oldalhosszának összege nagyobb a harmadik oldal hosszánál.

```
program haromszog;
var
  a, b, c, s, ter, ker : real;
begin
  writeln('A háromszög oldalai: ');
  write('a oldal: '); readln(a);
  write('b oldal: '); readln(b);
  write('c oldal: '); readln(c);
  if (a > 0) and (b > 0) and (c > 0) and
     (a + b > c) and (b + c > a) and (a + c > b) then
  begin
    s := (a + b + c)/2;
    ker:= a+b+c;
    ter:= sqrt(s*(s-a)*(s-b)*(s-c));
    writeln('Terület: ',ter:6:2);
    writeln('Kerület: ',ker:6:2);
  end
  else
    writeln('A háromszög nem szerkeszthető meg.');
  readln;
end.
```

*A program futásainak eredményei:*

```
A háromszög oldalai:
a oldal: 1
b oldal: 2
c oldal: 3
A háromszög nem szerkeszthető meg.

A háromszög oldalai:
a oldal: 3
b oldal: 4
c oldal: 5
Terület:   6.00
Kerület:  12.00
```

### *6.2.2.2 A case utasítás*

A **case** utasítás segítségével könnyen megoldhatjuk programunk, kifejezés értékétől függő, többirányú elágaztatását.

```
case kifejezés of
   érték1 :  utasítás1;
   érték2 :  utasítás2;
   ...
   értékN :  utasításN;
   else
     különben_utasítás;
end;
```

A **case** foglalt szó után csak sorszámozott típusú (***integer***, ***char***, ***boolean*** stb.) *kifejezés* állhat. A *kifejezés* értékétől függően történik a program következő utasításának (*utasításX*) a kiválasztása, az előtte álló sorszámozott típusú konstans (*értékX*) értéke alapján. Amennyiben a *kifejezés* értéke egyik konstanssal sem egyezik, akkor az **else**-ágra adódik a vezérlés. Az adott ágban szereplő utasítás végrehajtását követően, a program a **case**-szerkezetet záró **end** után álló utasítással folytatódik. Ugyancsak ide kerül a vezérlés, ha nem adjuk meg az **else**-ágat, és egyik konstanssal sem egyezik meg a kifejezés értéke.

Néhány megjegyzés az utasítással kapcsolatban:

- Az esetkiválasztó konstansok helyén konstans (fordítói) kifejezés, vesszővel tagolt konstanssorozat és résztartomány (például *12..23*) is szerepelhet.

```
var
  ch : char;
begin
  case ch of
     'I' : writeln('Iván');
     'N' : writeln('Natália');
     chr(ord('A')+25) : writeln('Zoltán');
     'P', 'S', 'K' : writeln('Péter, Sára, Klára');
     'A'..'C' : writeln('Anna, Bence, Cecília');
  end;
  readln;
end.
```

- A konstansok típusának egyeznie kell a **case** után álló kifejezés típusával.
- Itt az **else** előtt állhat pontosvessző, azonban használata nem kötelező.
- Free Pascalban az esetkonstansok között nem lehet átfedés.

Az elmondottak jobb megértését segítik a következő példaprogramok.

Készítsünk programot, amely beolvassa az írásbeli vizsga pontszámát, a pontszám alapján egész számú érdemjeggyel osztályoz, majd pedig az osztályzat alapján szöveges formában is megjeleníti a vizsga eredményét! (*vizsga*)

```
program vizsga;
var
  pontszam, erdemjegy : integer;
begin
  write('Pontszám: '); readln(pontszam);
  if (pontszam >=0) and (pontszam <=100) then
  begin
    if (pontszam >= 85) then erdemjegy := 5
    else if (pontszam >= 70) then erdemjegy := 4
    else if (pontszam >= 55) then erdemjegy := 3
    else if (pontszam >= 40) then erdemjegy := 2
    else erdemjegy := 1;
    write('Érdemjegy: ',erdemjegy);
    case erdemjegy of
       1: writeln(' (elégtelen)');
       2: writeln(' (elégséges)');
       3: writeln(' (közepes)');
       4: writeln(' (jó)');
       5: writeln(' (jeles)');
    end
  end
  else
    writeln('Hibás adat!');
  readln;
end.
```

*A program futásainak eredménye:*

```
Pontszám: 79
Érdemjegy: 4 (jó)

Pontszám: 89
Érdemjegy: 5 (jeles)

Pontszám: 68
Érdemjegy: 3 (közepes)

Pontszám: 45
Érdemjegy: 2 (elégséges)

Pontszám: 23
Érdemjegy: 1 (elégtelen)
```

Írjunk programot, amely beolvas két valós számot és egy karaktert! Amennyiben a karakter a négy alapművelet valamelyikének jele, elvégzi a kijelölt műveletet és megjeleníti az eredményt! (*kalkulator*)

```
program kalkulator;
var
  adat1, adat2, eredmeny: real;
  muvjel: char;
  hiba : integer;
begin
  write('1. adat: '); readln(adat1);
  write('2. adat: '); readln(adat2);
  hiba:=0;
  write('Műveleti jel: '); readln(muvjel);
  case muvjel of
    '+': eredmeny := adat1 + adat2;
    '-': eredmeny := adat1 - adat2;
    '*': eredmeny := adat1 * adat2;
    '/': begin
           if adat2 = 0 then
             begin
               eredmeny:=0; hiba:= 1;
               writeln('Az osztandó zérus!');
             end
           else
             eredmeny := adat1 / adat2;
         end;
    else
         begin
           writeln('Hibás művelet');
           hiba:= 1;
         end;
  end;
  if hiba = 0 then
   begin
     writeln('A művelet eredménye:');
     writeln(adat2:6:2,' ',muvjel, ' ', adat2:6:2,' = ',eredmeny:6:2);
   end;
  readln;
end.
```

*A program futásainak eredménye:*

```
1. adat: 3
2. adat: 5
Műveleti jel: +
A művelet eredménye:
  3.00 +   5.00 =   8.00

1. adat: 2
2. adat: 4
Műveleti jel: *
A művelet eredménye:
  2.00 *   4.00 =   8.00

1. adat: 7
2. adat: 3
Műveleti jel: /
A művelet eredménye:
  7.00 /   3.00 =   2.33
```

```
1. adat: 2
2. adat: 0
Műveleti jel: /
Az osztandó zérus!
```

### 6.2.3 Ciklusutasítások - iteráció

Algoritmusok egy részénél, az eredmény elérése érdekében, egy adott tevékenységet egymás után többször is végre kell hajtani (iteráció). Az ilyen ismétlődő tevékenységek Pascal nyelven történő megfogalmazására használjuk a ciklusutasításokat.

A Free Pascal az iteráció megvalósítását három utasítással támogatja. Ha a ciklusba való belépés előtt ismerjük az ismétlések számát, akkor általában a **for** utasítást használjuk. Amennyiben az ismétlések száma attól függ, hogy egy feltétel mikor válik hamissá (igazzá), akkor a **while (repeat-until)** utasítást alkalmazzuk. Ha sohasem áll elő a kilépés feltétele, akkor ún. végtelen ciklust kapunk, ami általában igen kellemetlen programozási hiba. A ciklusban ismétlődő tevékenységet (tevékenységsorozatot) szokás ciklusmagnak nevezni.

A ciklusokat szokás a belépési/kilépési feltétel ellenőrzésének helye szerint is csoportosítani. A **for** és a **while** a ciklusmag végrehajtása előtt (elöltesztelő ciklus), míg a **repeat-until** a ciklusmag lefutása után ellenőrzi a feltételt (hátultesztelő ciklus). Alapvető különbség a két csoport között, hogy a hátultesztelő ciklus magja legalább egyszer mindig végrehajtódik.

#### *6.2.3.1 A for utasítás*

A **for** ciklusutasítást akkor használjuk, ha valamely sorszámozott típus értékkészletének résztartományán kívánunk végiglépkedni. Ezt megtehetjük a kisebb értékektől a nagyobbak felé (**to**), illetve fordítva, a csökkenő értékek irányába (**downto**) haladva. A két esetnek megfelelően az utasítás két formája:

```
for ciklusváltozó:=kezdőérték to végérték do utasítás;

for ciklusváltozó:=kezdőérték downto végérték do utasítás;
```

A sorszámozott típus folytonos résztartományát a *kezdőérték* és a *végérték* kifejezések jelölik ki. A ciklus futása során, az ugyancsak sorszámozott típusú *ciklusváltozó* automatikusan felveszi az intervallum soron következő értékét. A **do** foglalt szó után megadott *utasítás* (amely lehet összetett utasítás is) alkotja a ciklusmagot, mely ismétlésének száma pontosan meghatározható:

```
|végérték-kezdőérték|+1
```

A **for** utasítás első formájában (az ún. előre haladó ciklusban) csak akkor hajtódnak végre a ciklusmag utasításai, ha a *kezdőérték* ≤ *végérték*. A *ciklusváltozó*, a *kezdőértékről* indulva minden lefutás után a ***succ***(*ciklusváltozó*) értéket veszi fel. (Ez a művelet egész típusok esetén eggyel való növelést jelent). A ciklus akkor fejezi be működését, ha a ***succ***(*ciklusváltozó*) kifejezés értéke túllépi a *végértéket*. A *ciklusváltozó* értékét a ciklusmagban felhasználhatjuk, azonban megváltoztatását a Free Pascal nem engedi. A ciklus lefutása után a *ciklusváltozó* értéke Free Pascalban megegyezik a *végértékkel*.

Gyakran használjuk a **for** ciklust egy tevékenység adott számszor történő végrehajtására. Például, az alábbi ciklus a ***writeln()*** eljárás hívását 12-szer ismételve megjeleníti az első 12 négyzetszámot:

```
var
  x : integer;
begin
  for x := 1 to 12 do
    writeln(sqr(x):5);
end.
```

A **for** utasítás visszafelé lépkedő (**downto**) formája csak akkor kezdi el a működését, ha teljesül a *kezdőérték* ≥ *végérték* feltétel. A *ciklusváltozó* ebben az esetben minden lefutás után a ***pred***(*ciklusváltozó*) függvény szerint változtatja értékét. (Ez a hívás egész típusok esetén eggyel való csökkentést jelent).

Az alábbi példában az angol ABC nagybetűit karaktersorozattá fűzzük össze, és minden lépésben megjelenítjük a sztringet:

```
var
  ch : char;
  s  : string;
begin
  s := '';
  for ch:='Z' downto 'A' do
    begin
      s := s + ch;
      writeln(s);
    end;
end.
```

Vannak esetek, amikor több ciklus egybeépítésével, egymásba ágyazásával (skatulyázásával) érjük el célunkat. Ekkor a belső ciklus teljes lefutását követi a külső ciklus léptetése.

Az alábbi példában a külső ciklus (*a*) kijelöli a táblázat sorait, míg a belső ciklus (*b*) feltölti azokat adatokkal:

```
var
  a, b : integer;
begin
  for a:=0 to 3 do
   begin
     for b:= 0 to a+1 do
        write(a+b : 5);
     writeln;
   end;
  readln;
end.
```

*A program futásának eredménye:*

```
    0    1
    1    2    3
    2    3    4    5
    3    4    5    6    7
```

Nézzünk néhány kidolgozott példaprogramot a **for** utasítás alkalmazására!

Készítsünk programot, amely kiszámítja az 1 és 12 közötti páros számok szorzatát, valamint a páratlanok összegét! (*paritasfor*)

```
program paritasfor;
var
  i: integer;
  osszeg  : longint = 0;
  szorzat : longint = 1;
begin
  for i:=1 to 12 do
    if i mod 2 = 0 then
      szorzat:=szorzat*i
    else
      osszeg:=osszeg+i;
  writeln('Páratlanok összege: ',osszeg);
  writeln('Párosak szorzata  : ',szorzat);
  readln;
end.
```

*A program futásának eredménye:*

```
Páratlanok összege: 36
Párosak szorzata  : 46080
```

Pascal program segítségével jelenítsük meg a prímszámokat adott határig! (A prímszámok csak az eggyel és önmagukkal osztható, pozitív egész számok.) (*primek*)

A program futásának gyorsítása érdekében kihasználtuk azt, hogy a számok prímtényezős felbontása szimmetrikus, például *6 = 2*3 = 3*2*. Ezért az osztók keresését elegendő a szám négyzetgyökéig elvégezni.

```
program primek;
var
  max, n, i : longint;
  prim_e : boolean;
begin
  write('Meddig :'); readln(max);
  for n:=2 to max do
    begin
      prim_e:= true;
      for i:=2 to round(sqrt(n)) do
         if n mod i = 0 then prim_e := false;
      if prim_e then write(n : 5);
   end;
  readln;
end.
```

*A program futásának eredménye:*

```
Meddig :23
    2    3    5    7   11   13   17   19   23
```

Írjunk programot, amely beolvassa a betét értékét, a kamat nagyságát, valamint az évek számát, melyre leköti a betétet. Számítsuk ki az adatok alapján a kamatos kamat értékét! (*kamatpr*)

```
program kamatpr;
var
  i, ev, kamat: integer;
  betet, toke, kamatok: real;
begin
  write('alap : '); readln(betet);
  write('kamat: '); readln(kamat);
  write('év   : '); readln(ev);
  toke:=betet;
  for i:=1 to ev do
    begin
     kamatok:=toke*kamat/100;
     toke:=toke+kamatok;
    end;
  writeln('kamatos kamattal növelt tőke: ',toke:12:1);
  readln;
end.
```

*A program futásának eredménye:*

```
alap : 1000
kamat: 10
év   : 10
kamatos kamattal növelt tőke:       2593.7
```

### *6.2.3.2 A while utasítás*

Ha ciklusszervezés során valamely feltételtől függően kell ismételnünk egy tevékenységet, akkor a feltételes ciklusutasítások egyikét kell használnunk. Az esetek többségében a programozási feladat bármelyikkel megoldható. Amennyiben a megadott utasítást legalább egyszer végre kell hajtani, akkor a **repeat-until** használata javasolt. Ellenkező esetben a **while** ciklussal érjük el célunkat, melynek általános formája:

```
while feltétel do utasítás;
```

A **while** ciklusban megadott *utasítás* csak akkor hajtódik végre, ha a (belépési) *feltétel* igaz értékű. Amikor a feltétel hamisra változik, a ciklus befejezi a működését, és a vezérlés átadódik a **while** ciklust követő utasításra.

Abban az esetben, ha a *feltétel* értéke a ciklusba való belépéskor ***false***, a ciklusmag egyszer sem hajtódik végre. Amennyiben a belépési *feltétel* ***true*** értéke nem változik meg a ciklus működése során, akkor végtelen ciklust kapunk. Végtelen ciklust akaratunk ellenére is kaphatunk, ha a **do** után pontosvesszőt (üres utasítást) teszünk:

```
i := 0;
while i<=23 do; // ← végtelen ciklus a pontosvessző miatt!
  inc(i);
```

Az alábbi programrészletben 1-től 20-ig összegezzük az egész számokat:

```
szam:=1;
osszeg:=0;
while szam < 21 do
  begin
    osszeg:=osszeg+szam;
    szam:=szam+1;
  end;
writeln('Az összeg while ciklussal: ', osszeg);
```

Az alábbi példaprogramok jól szemléltetik a **while** ciklus használatát.

Készítsünk programot, amely kiszámítja az 1 és 12 közötti páros számok szorzatát, valamint a páratlanok összegét! (*paritaswhile*)

```
program paritaswhile;
var
  i: integer;
  osszeg  : longint = 0;
  szorzat : longint = 1;
begin
  i := 1;
  while i <= 12 do
    begin
```

```
      if i mod 2 = 0 then
         szorzat:=szorzat*i
      else
         osszeg:=osszeg+i;
      i := i+1;
    end;
  writeln('Páratlanok összege: ',osszeg);
  writeln('Párosak szorzata  : ',szorzat);
  readln;
end.
```

*A program futásának eredménye:*

```
Páratlanok összege: 36
Párosak szorzata  : 46080
```

Írjunk programot, amely pont jelig olvas karaktereket! Számláljuk meg a szóközök, az *a* és az *e* betűk számát! (*betuszamlalo*)

```
program betuszamlalo;
var
  adb, edb, spdb: integer;
  c: char;
begin
  adb:=0; edb:=0; spdb:=0;
  c:=#32;  // szóköz
  writeln('Karaktereket olvas a pontig!');
  while c <> '.' do
    begin
      read(c);
      case c of
        'a': inc(adb);
        'e': inc(edb);
        ' ': inc(spdb);
       end;
    end;
  writeln('az a karakterek száma : ',adb);
  writeln('az e karakterek száma : ',edb);
  writeln('a szóközök száma      : ',spdb);
  readln; readln;
end.
```

*A program futásának eredménye:*

```
Karaktereket olvas a pontig!
ezeket a karaktereket a pont zárja.
az a karakterek száma : 5
az e karakterek száma : 6
a szóközök száma      : 5
```

Készítsünk programot, amely megjeleníti a beolvasott egész szám prímtényezős felbontását! (*primtenyezok*)

```
program primtenyezok;
var
  szam, n : longint;
  prim : boolean;
begin
  write('Kérek egy egész számot: '); readln(szam);
  write(szam:6,' = 1');
  prim := true;
  for n:=2 to szam do
    while szam mod n = 0 do
      begin
        write('*',n);
        szam := szam div n;
        prim := false;
      end;
  if prim then write('*',szam);
  readln;
end.
```

*A program futásainak eredménye:*

```
Kérek egy egész számot: 12
    12 = 1*2*2*3

Kérek egy egész számot: 23
    23 = 1*23

Kérek egy egész számot: 2006
  2006 = 1*2*17*59
```

### *6.2.3.3 A repeat-until utasítás*

A feltételes ciklusutasítások másik képviselője a Pascal nyelvben a **repeat-until** ciklus, melynek általános formája:

```
repeat
    utasítás1;
...
    utasításN;
until feltétel;
```

A **repeat** és **until** foglalt szavak között megadott utasítások legalább egyszer mindig végrehajtódnak, mivel a *feltétel* vizsgálata a ciklus végén áll. A ciklus akkor fejezi be működését, ha a (kilépési) *feltétel* ***true*** értékűvé válik.

Nézzünk néhány példát a **repeat-until** ciklus működésének jobb megértése érdekében!

Készítsünk programot, amely kiszámítja az 1 és 12 közötti páros számok szorzatát, valamint a páratlanok összegét! (*paritasrepeat*)

```
program paritasrepeat;
var
  i: integer;
  osszeg  : longint = 0;
  szorzat : longint = 1;
begin
  i := 1;
  repeat
    if i mod 2 = 0 then
       szorzat:=szorzat*i
    else
       osszeg:=osszeg+i;
    i := i+1;
  until i > 12;
  writeln('Páratlanok összege: ',osszeg);
  writeln('Párosak szorzata  : ',szorzat);
  readln;
end.
```

*A program futásának eredménye:*

```
Páratlanok összege: 36
Párosak szorzata  : 46080
```

A *paritaswhile* programmal összevetve a megoldást, jól látható, hogy az azonos tevékenységet végző **while** és **repeat-until** ciklusokban használt feltételek egymás ellentettei.

Írjunk programot, amely az *Enter* billentyű leütéséig olvas karaktereket, és megszámlálja a beadott szövegben az angol magán- és mássalhangzók, valamit az írásjelek számát! (*betuosztalyozo*)

```
program betuosztalyozo;
var
  mgh, msh, irasjel : integer;
  c: char;
begin
  mgh:=0; msh:=0; irasjel:=0;
  writeln('Karaktereket olvas az Enter leütéséig!');
  repeat
      read(c);
      case upcase(c) of
        'A','E','I','O','U': inc(mgh);
        'B'..'D', 'F'..'H', 'J'..'N', 'P'..'T', 'V'..'Z' : inc(msh);
        ',','.','!','?',':',';' : inc(irasjel);
       end;
  until c = #10;
  writeln('magánhangzók száma  : ',mgh);
  writeln('mássalhangzók száma : ',msh);
  writeln('írásjelek           : ',irasjel);
  readln;
end.
```

*A program futásának eredménye:*

```
Karaktereket olvas az Enter leütéséig!
abcdefghijklmnopqrstuvwxyz.,:;?!
magánhangzók száma   : 5
mássalhangzók száma  : 21
írásjelek            : 6
```

### *6.3.2.4 A break és a continue foglalt szavak*

A Free Pascal nyelv ciklusainak szervezésekor problémát okoz, ha a ciklusból, adott feltétel teljesülése esetén (a ciklus lefutása előtt) ki kell lépni. Ugyancsak nehézségekbe ütközünk, ha a ciklusmag hátralevő utasításait át kell ugranunk. Az ilyen és hasonló esetekben ugyan használhatjuk a **goto** utasítást, azonban sokkal biztonságosabb megoldáshoz jutunk a **break**, illetve a **continue** foglalt szavak alkalmazásával.

Meg kell jegyeznünk, hogy a **break** és a **continue** foglalt szavak is (helyi) vezérlésátadást hajtanak végre, mely az esetek többségében a program átszervezésével kiküszöbölhető. A ***break*** jól használható keresést végző ciklusokban, ahol találat esetén kiugrunk a ciklusból.

Nézzünk néhány példát a **break** és a **continue** alkalmazására, illetve a használatuk elkerülésére!

Készítsünk programot, amely a megadott egész számról megmondja, hogy prímszám-e! Használjuk a **break** foglalt szót! (*primszame1*)

```
program primszame1;
var
   i, n : longint;
   prim : boolean = true;

begin
   write('Kérek egy számot: ');
   readln(n);
   for i:=2 to round(sqrt(n)) do
     if n mod i =0 then
       begin
          prim := false;
          break;
       end;

   case prim of
     true  : writeln('prím');
     false : writeln('nem prím');
   end;
   readln;
end.
```

*A program futásainak eredménye:*

```
Kérek egy számot: 12
nem prím

Kérek egy számot: 23
prím
```

Készítsünk **break** nélküli programot, amely a megadott egész számról megmondja, hogy prímszám-e! (*primszame2*)

```
program primszame2;
var
   i, n : longint;
   prim : boolean = true;
begin
   write('Kérek egy számot:');
   readln(n);
   i:=2;
   while (i<=round(sqrt(n))) and prim do
     begin
       if n mod i =0 then
          prim := false;
       inc(i);
     end;
   case prim of
     true  : writeln('prím');
     false : writeln('nem prím');
   end;
   readln;
end.
```

*A program futásainak eredménye:*

```
Kérek egy számot: 12
nem prím

Kérek egy számot: 23
prím
```

Írjunk programot, amely az első, megadott darabszámú páros számot összegzi! Használjuk a **for** ciklust és a **continue** foglalt szót! (*sorosszeg1*)

```
program sorosszeg1;
var  s : longint;
     n, i : integer;
begin
   write('A számsor elemeinek száma: ');
   readln(n);
   for i:=1 to 2*n do
     begin
       if i mod 2 <> 0 then continue;
       s := s + i;
     end;
```

```
   writeln('A sorösszeg: ', s);
   readln;
end.
```

*A program futásának eredménye:*

```
A számsor elemeinek száma:7
A sorösszeg: 56
```

Készítsük el a *sorosszeg1* példaprogram **continue** foglalt szó nélküli változatát! (*sorosszeg2*)

```
program sorosszeg2;
var  s : longint;
     n, i : integer;
begin
   write('A számsor elemeinek száma: ');
   readln(n);
   for i:=1 to 2*n do
     if i mod 2 = 0 then s := s + i;
   writeln('A sorösszeg: ', s);
   readln;
end.
```

*A program futásának eredménye:*

```
A számsor elemeinek száma:7
A sorösszeg: 56
```

### 6.2.4 A with utasítás

A **with** utasítás megkönnyíti a rekordváltozók és az objektumok elemeire (mezőire, metódusaira) való hivatkozást. Az utasítás általános formája:

**with** *rekordváltozó/objektum* **do** *utasítás;*

A **record**, **object** és **class** típusú adatokkal könyvünk további fejezeteiben foglalkozunk.

### 6.2.5 A kivételkezelés utasításai

A Free Pascal ***ObjFPC*** és ***Delphi*** módjaiban működő kivételkezelés lehetővé teszi, hogy strukturált módon készüljünk fel a program futását megszakító hibák, illetve más kivételes események kezelésére. A programunkban megjelölhetjük a „veszélyes" részeket,

**try** ... **except** ... **end;**  **try** ... **finally** ... **end;**

és megmondhatjuk, hogy mi történjen a különböző hibák esetén. A **raise** utasítást segítségével a kivételeket magunk is előidézhetjük. A kivételkezelés fogásait a későbbiekben részletesen ismertetjük.

# 7. Strukturált típusok

A strukturált (összetett) típusok közös jellemzője, hogy összetevőkből épülnek fel. A strukturált típusú változók összetevői önállóan is elérhetők, azonban a teljes szerkezet egyben is kezelhető. A Free Pascal strukturált típusai:

- tömb (**array**),
- rekord (**record**),
- halmaz (**set**),
- állomány (**file**),
- objektum (**object**),
- osztály (**class**), interfész (**interface**) (*ObjFPC* és *Delphi* módokban).

Free Pascalban a fenti típusok közül az **array**, **record**, **object** és **class** típusok esetén a **packed** módosítóval, illetve a *$PACKRECORDS* fordítási direktívával szabályozhatjuk, hogy a típust alkotó adatelemek milyen memóriahatáron kezdődjenek. Alapértelmezés szerint az elemek szóhatáron (páros tárcímen) helyezkednek el. A **packed** előírással az összetevők bájthatárra igazítását írjuk elő, míg a *$PACKRECORDS* fordítási direktívával 1, 2, 4, 8 vagy 16 bájtos határt is megadhatunk. 32-bites rendszerekben a memória gyorsabb elérését a duplaszó (4-bájtos), vagy annál nagyobb értékű kiigazítás biztosítja, azonban kisebb értékek esetén tárhelyet takarítunk meg. Az alábbi példában mindkét módszerrel a bájtos kiigazítást kérjük a fordítótól:

```
type datum =packed record
        ev  : word;
        ho  : 1..12;
        nap : 1..31;
     end;
```

```
{$PACKRECORDS 1}
type datum = record
        ev  : word;
        ho  : 1..12;
        nap : 1..31;
     end;
{$PACKRECORDS Default}
```

A strukturált típus, illetve változó számára kijelölt tárterület méretét a *sizeOf()* függvénnyel tudhatjuk meg, melynek argumentuma a típus, illetve a változó neve.

## 7.1 Tömbtípusok

A tömböket adott darabszámú, azonos típusú adatok (elemek) tárolására használjuk. A tömbtípus deklarálásának általános formája:

```
type
   tömbtípus = ⟨packed⟩ array [sorszámozott_típus] of elemtípus;
```

A deklarációban a **packed** szó megadása opcionális, ezt jelölik a ⟨⟩ zárójelek. A szögletes zárójelek között megadott típussal az indexek rendezett értékkészletét adjuk meg (ami általában résztartomány). A típusleírásban az **of** szó utáni elemtípus megha-

tározza, hogy a tömb elemei milyen adatokat tárolnak. Az elemek típusa egyaránt lehet egyszerű vagy strukturált típus.

A fentiek szerint definiált tömbök ***statikusak***, mivel a méretük nem változtatható meg a deklarációt követően, csak a tartalmuk. Könyvünk 10.4. fejezetében foglaltuk össze a ***dinamikusan*** változtatható méretű tömbök használatával kapcsolatos ismereteket. Jelen fejezetben kizárólag a statikus tömbökkel foglalkozunk, amelyek a memóriában mindig folytonos területen helyezkednek el.

A tömbök lehetnek egy-, két- vagy többkiterjedésűek (dimenziósak). A deklarációban a szögletes zárójelek között annyi indextípust kell veszővel elválasztva felsorolni, ahány dimenziós tömböt kívánunk létrehozni. Az egydimenziós tömböt a matematikában vektornak, míg a kétdimenziósat mátrixnak nevezzük. Példaként nézzünk néhány tömbdefiníciót!

```
var
   t1 : array[1..12] of double;
   t2 : array['A'..'Z'] of integer;
   t3 : array[-5..5, 12..23] of byte;
   t4 : array[boolean, char] of integer;
   t5 : array[boolean] of array[char] of integer;
```

A *t4* és a *t5* tömbök azonos típusúak a Pascalban, mivel a kétdimenziós tömböket egydimenziós tömbök tömbjeként értelmezi a fordító.

A tömbök elemeire a tömbváltozó nevével és egy indexszel (indexeszekkel) hivatkozunk. A szögletes zárójelek között megadott index sorszámozott típusú változó, konstans vagy kifejezés lehet.

```
t1[1], t1[2], ... , t1[12]          t2['A'], t2['B'], ... , t2['Z']
```

Többdimenziós tömbök esetén kétféleképpen is elérhetjük az elemeket.

```
t3[1,12]              vagy          t3[1][12]
t4[false,'N']         vagy          t4[false]['N']
t5[false,'N']         vagy          t5[false]['N']
```

Az indexhatár átlépése igen gyakran előforduló programozási hiba. Ez a hiba azonban kideríthető, ha bekapcsoljuk a tartományhatárok figyelését, a *{$RANGECHECKS ON}* fordítási direktíva megadásával. (Alaphelyzetben a Free Pascal fordító nem ellenőrzi a tartományhatárok átlépését.)

A tömbökről információkat szerezhetünk a már megismert ***low()***, ***high()***, ***sizeOf()*** és ***length()*** függvényekkel. Tömb esetén a ***low()*** az első index kezdőértékét, míg a ***high()*** a végértékét adja vissza. Többdimenziós esetben, az alábbi példában szereplő módon kérdezhetjük le az második kiterjedéshez tartozó indexhatárokat:

```
var
  X : packed array[12..23, 7..17] of integer;
begin
  writeln( low(X) );              //12
  writeln( high(X ));             //23
  writeln( low(X[low(X)]) );      // 7
  writeln( high(X[low(X)]) );     //17
  writeln( sizeof(X) );           //a tömb mérete: 264 bájt
  writeln( sizeof(X[low(X)]) ); //a tömb sorainak mérete: 22 bájt
  writeln( length(X) );           //a tömb sorainak száma: 12
  writeln( length(X[low(X)]) ); //a tömb oszlopainak száma: 11
end.
```

### 7.1.1 Művelet azonos típusú tömbökkel

A strukturált típusokkal kapcsolatosan fontos tisztázni az azonos típus fogalmát. Az alábbi példában, a Free Pascal – ellentétben a szabványos Pascal nyelvvel – mindhárom tömböt azonos típusúnak tekinti:

```
var
  t1,t2 : array [1..12] of integer;
  t3    : array [1..12] of integer;
begin
  t1:= t2;
  t1:= t3;
end.
```

Ennek ellenére javasolt, hogy a tömb típusát névvel lássuk el, így minden Pascal változatban azonos típusú tömböket hozhatunk létre:

```
type
  ttomb = array [1..12] of integer;
var
  t1, t2, t3 : ttomb;
begin
  t1:= t2;
  t1:= t3;
  readln;
end.
```

A típusazonosság fontos szerepet játszik a tömbök közötti értékadás művelete esetén, illetve az alprogramok tömbparaméterrel történő hívásakor. Amennyiben a *t1* és *t3* azonos típusú tömbök, az alábbi értékadás során a *t3* tartalma átmásolódik a *t1*-be:

```
t1:=t3;
```

### 7.1.2 Tömbök kezdőértéke

Free Pascalban, az egyszerű típusú változókhoz hasonlóan, a tömbtípusú konstansokat és változókat is elláthatjuk kezdőértékkel. Egydimenziós tömb esetén kerek zárójelek

között, vesszővel elválasztva, pontosan annyi elemet kell felsorolunk, amennyi a tömb elemeinek száma.

```
const
  szamok : array[1..5] of integer = (1, 5, 12, 34, 56);
  sz1    : array[1..8] of char = 'Eredmény';
  sz2    : array[1..8] of char =('E','r','e','d','m','é','n','y');
var
  sz3   :array[boolean] of string = ('nem','igen');
```

Kétdimenziós tömbök esetén külön zárójelezzük az eggyel kisebb dimenzióhoz (sorokhoz) tartozó kezdőértékeket, mivel a rendszer a kétdimenziós tömböket sorfolytonosan tárolja:

```
type
  tomb = array[1..2,1..3] of integer;
const
  adat : tomb = ((3,1,5),(2,8,4));
```

### 7.1.3 Tömbkezelési példaprogramok

Nézzünk néhány, a tömbök használatát szemléltető, példaprogramot! Érdemes megfigyelni, hogy a tömb elemeit a **for** ciklus segítségével kényelmesen és biztonságosan elérhetjük.

Írjunk programot, amely először beolvassa az egész adatok darabszámát, majd pedig bekéri az adatokat egy tömbbe, végül pedig kiszámítja az elemek összegét! A program legfeljebb 100 számot tudjon feldolgozni! (*tombolv*)

```
program tombolv;
const
    maxn = 100;
var
   i, n : integer;
   x : array[1.. maxn] of integer;
   osszeg : integer;
begin
  repeat
     write('Adatok száma: '); readln(n);
  until (n>0) and (n<=maxn);
  for i:=1 to n do
  begin
    write('x[',i:2,'] = '); readln(x[i]);
  end;
  osszeg := 0;
  for i:=1 to n do
    osszeg := osszeg + x[i];
  writeln('A tömb adatainak összege : ',osszeg);
  readln;
end.
```

*A program futásának eredménye:*

```
Adatok száma: 0
Adatok száma: 123
Adatok száma: 4
x[ 1] = 12
x[ 2] = 23
x[ 3] = 7
x[ 4] = 26
A tömb adatainak összege : 68
```

Készítsünk programot, amely egy legfeljebb 10x10-es négyzetes mátrixot véletlenszerűen feltölt bájtértékekkel, majd mátrixos formában megjeleníti az elemeket! (*matrix*)

```
program matrix;
const
   maxn = 10;
type
   tmat = array[1..maxn, 1..maxn] of byte;
var
   i,j,n : integer;
   m : tmat;
begin
  repeat
     write('A négyzetes mátrix mérete: '); readln(n);
  until (n>1) and (n<=maxn);
  randomize;
  for i:=1 to n do
    for j:=1 to n do
      m[i,j] := random(256);
  for i:=1 to n do
    begin
       for j:=1 to n do
          write(m[i,j]:4);
       writeln;
    end;
  readln;
end.
```

*A program futásának eredménye:*

```
A négyzetes mátrix mérete: 5
  16 119 226 110 138
 130 131 215 236 236
 166 240  19 108  81
  79  38 235 249 219
 172 126 186 237 250
```

Írjunk programot, amely beolvas egy szöveget, és statisztikát készít a szövegben előforduló angol ABC betűiről! (*chstat*)

```
program chstat;
type
   tabc = array['A'..'Z'] of byte;
var
   ci : char;
   s : tabc;
   x : string;
   i : integer;
begin
  // A statisztika tömb elemeinek nullázása
  for ci:='A' to 'Z' do
    s[ci] := 0;
  write('Kérek egy szöveget: '); readln(x);
  for i:=1 to length(x) do // csak betűk esetén léptetünk
    if x[i] in ['a'..'z','A'..'Z'] then inc(s[upcase(x[i])]);
  // Csak az előforduló karaktereket írjuk ki:
  for ci:='A' to 'Z' do
    if s[ci]<>0 then
      writeln(ci:2,#32,s[ci]:5);
  readln;
end.
```

*A program futásának eredménye:*

```
A Kérek egy szöveget: Indul a gorog aludni
 A     2
 D     2
 G     2
 I     2
 L     2
 N     2
 O     2
 R     1
 U     2
```

Készítsünk programot, amely konstans tömbbel inicializál egy normál vektort, majd növekvő sorrendbe rendezi annak elemeit! (*rendezes*)

```
program rendezes;
type
   tvekt = array[1..10] of integer;
const
   ct : tvekt =(7,12,23,35,89,23,19,4,9,60);
var
   t : tvekt;
   i, j, e : integer;

begin
  t := ct;
  write('rendezetlen:');
  for i:=low(t) to high(t) do
    write(t[i]:5);
  writeln;
```

```
  // Rendezés
  for i:=low(t) to high(t)-1 do
     for j:=i+1 to high(t) do
        if t[i]>t[j] then  // ha kell cserélünk
          begin
             e    := t[i];
             t[i] := t[j];
             t[j] := e;
          end;
  write('rendezett  :');
  for i:=low(t) to high(t) do
    write(t[i]:5);
  writeln;
  readln;
end.
```

*A program futásának eredménye:*

```
rendezetlen:     7    12    23    35    89    23    19     4     9    60
rendezett  :     4     7     9    12    19    23    23    35    60    89
```

## 7.2 Rekordtípusok

A rekord olyan összetett adatszerkezet, amelyben tetszőleges számú, azonos vagy különböző típusú elemet (adatmezőt) fogunk csokorba. A rekorddeklaráció általános formája:

```
type
   rekordtípus = ⟨packed⟩ record
      mező1: típus1;
     ...
      mezőN: típusN;
   end;
```

A mezőket a közönséges változókhoz hasonlóan adjuk meg, azonban nem tartalmazhatnak kezdőértéket. Példaként tekintsük a fejezet bevezetőjében szereplő *datum* adattípust!

```
type
    datum = record
               ev  : integer;
               ho  : 1..12;
               nap : 1..31;
            end;
```

Amennyiben naptárt kívánunk készíteni, egy adott naphoz tartozó bejegyzést az alábbi rekordban tárolhatjuk, melyet kétféleképpen is létrehozhatunk:

```
type
    naptar1 = record
        evnap : datum;
        bejegyzes : string[128];
    end;
```

```
type
    naptar2 = record
        evnap : record
                  ev  : integer;
                  ho  : 1..12;
                  nap : 1..31;
                end;
        bejegyzes : string[128];
    end;
```

### 7.2.1 A rekordok kezelése - a with utasítás

A fentiek közül bármelyik típussal is hozzuk létre a *ma* változót, az alábbi értékadásokkal adhatunk értéket a mezőinek:

```
var
   ma, holnap : naptar1;
begin
  ma.evnap.ev:=2006;
  ma.evnap.ho:=10;
  ma.evnap.nap:=2;
  ma.bejegyzes := 'Születésnap';
  with ma do
    begin
      evnap.ev:=2006;
      evnap.ho:=10;
      evnap.nap:=2;
      bejegyzes := 'Születésnap';
    end;
  with ma do
    begin
      with evnap do
        begin
          ev:=2006;
          ho:=10;
          nap:=2;
        end;
      bejegyzes := 'Születésnap';
    end;
  with ma, evnap do
    begin
      ev:=2006;
      ho:=10;
      nap:=2;
      bejegyzes := 'Születésnap';
    end;
end.
```

A példaprogramból is látható, hogy a rekordok mezőire úgy hivatkozunk, hogy a rekordváltozó neve után ponttal elválasztva adjuk meg a mező azonosítóját:

```
ma.bejegyzes := 'Mai nap';          ma.evnap.ev := 1979;
```

Egy másik – kimondottan a Pascal nyelvre jellemző – lehetőség, a **with** utasítás használata. A **with** utasítás általános formája:

```
with rekordváltozó do utasítás;
```

A *rekordváltozó.mezőnév* hivatkozás helyett a **with** utasításon belül elegendő csak a mezőnevet megadni. A mezők ekkor automatikusan az utasításban szereplő *rekordváltozóhoz* tartoznak.

A példában az utasítás másik formáját is használjuk, amikor több név szerepel a *rekordváltozó* helyén:

```
with rekord1, rekord2, rekord3 do utasítás;
```

Ez a forma egymásba ágyazott **with** utasítások sorozatának felel meg:

```
with rekord1 do
   with rekord2 do
      with rekord3 do utasítás;
```

Két, azonos típusú rekordváltozó közötti értékadás elvégezhető mezőnként, de egyetlen lépésben is.

```
holnap.evnap.ev  := ma.evnap.ev;        holnap := ma;
holnap.evnap.ho  := ma.evnap.ho;
holnap.evnap.nap := ma.evnap.nap;
holnap.bejegyzes := ma.bejegyzes;
```

### 7.2.2 Variálható rekord

A rekordtípus használhatóságát növeli, hogy a rekordoknak, egy megadott feltételtől függően, különböző mezői is lehetnek. Az ilyen rekordot variálható rekordnak nevezzük. A variálható rekordban a lehetséges szerkezeteket a **case** foglalt szó mögött adjuk meg, a hagyományos mezők után:

```
type
   variálható_rekord = ⟨packed⟩ record
      mező1: típus1;
      ...
      mezőN: típusN;
      case ⟨esetmező:⟩ sorszámozott_típus of
         konstans1: (mező11: típus11; mező21: típus21; ⟨....⟩);
         ...
         konstansM: (mezőn1M: típus1M; mező2M: típus2M; ⟨....⟩);
   end;
```

A **case** kulcsszó után a *esetmező* elhagyható, ha azonban használjuk, akkor a rekord önálló mezőjeként kezelhető. A fordító a memóriát mindig a legnagyobb tárigényű esethez foglalja le, a variálhatóság csak a programozó kényelmét szolgálja. Az egyes esetekhez tartozó mezőneveknek – a rekordon belül – nem szabad ismétlődniük.

Az alábbi példában a térbeli pontokat kétféleképpen is megadhatjuk. A *polar* esetmezőt csak arra használjuk, hogy legyen információnk a tárolt adatok formátumáról:

```
type
  Pont3D = packed record
      case polar : boolean of
          false : (x, y, z : real);
          true :  (r, alfa, beta : real);
  end;
var
  p1, p2 : Pont3D;

begin
  with p1 do
    begin
      polar:=false;
      x:=1;
      y:=2;
      z:=3;
    end;
  with p2 do
    begin
      polar:=true;
      r:=123;
      alfa:=30;
      beta:=45;
    end;
  if p2.polar then
    with p2 do
      writeln(r:6:2, alfa:6:2, beta:6:2);
  readln;
end.
```

### 7.2.3 Rekord típusú konstansok

Külön kell szólnunk a rekordtípusú konstansok készítéséről, illetve a rekordváltozók kezdőértékkel való ellátásáról, mivel elég egyedi megadási módot kell használnunk.

```
type
    datum = record
                ev  : integer;
                ho  : 1..12;
                nap : 1..31;
            end;

    naptar = record
        evnap : datum;
        bejegyzes : string[128];
    end;
```

A fenti típusok segítségével készítsünk konstansokat! A konstansok inicializációs listájában a mezőnevet kettősponttal elválasztva követi a mező értéke, pontosvesszővel

lezárva. Az inicializációs listában a mezőnevek sorrendjének követnie kell a rekorddefinícióban szereplő sorrendet.

```
const
   karacsony : datum = (ev:2006; ho:12; nap:25);
   tanev : naptar =  (evnap:(ev:2006; ho:09; nap:1);
                      bejegyzes:'Évnyitó');

var
   mainap : naptar = (evnap:(ev:2006; ho:05; nap:23);
                      bejegyzes:'Félév');
```

### 7.2.4 Rekordelemű tömbök

A különböző strukturált típusokat egymásba is építhetjük. Készíthetünk például olyan rekordot, amelyben a mezők tömbök, de olyan tömb is kialakítható, amelyben az elemek rekordok. Ez utóbbira példa, az éves naptár kialakítása:

```
type
    datum = record
               ev  : integer;
               ho  : 1..12;
               nap : 1..31;
            end;
    naptar = record
        evnap : datum;
        bejegyzes : string[128];
    end;

const
   karacsony : datum = (ev:2006; ho:12; nap:25);
type
   evesnaptar = array[1..365] of naptar;
var
   naptar2006 : evesnaptar;
   i : integer;

begin
   for i:=1 to 365 do
     with naptar2006[i], evnap do
       begin
         ev:=2006;
         bejegyzes:='';
       end;

  naptar2006[1].evnap.ho :=1;
  naptar2006[1].evnap.nap:=1;
  naptar2006[1].bejegyzes:='Újév';

  naptar2006[359].evnap:=karacsony;
  naptar2006[359].bejegyzes:='Karácsony első napja';
end.
```

### 7.2.5 A rekordok használatát segítő példaprogramok

A rekord típusú változóknak az értékadó utasításban, egyetlen lépésben, illetve mezőnként adhatunk értéket. Ezzel szemben a mezőadatok felhasználótól való bekérését, illetve a képernyőn való megjelenítését csak mezőnként végezhetjük el.

Írjunk programot, amely egy *teglalap_tipus* típusú rekordba beolvassa a téglalap két oldalának hosszát, és kiszámítja a téglalap kerületét és területét! (*teglalappr*)

```
program teglalappr;
type
  teglalap_tipus = record
                     a_oldal, b_oldal: real;
                     ter, ker: real;
                   end;
var
    teglalap: teglalap_tipus;
begin
   write('Téglalap a oldala: '); readln(teglalap.a_oldal);
   write('Téglalap b oldala: '); readln(teglalap.b_oldal);
   teglalap.ter:=teglalap.a_oldal*teglalap.b_oldal;
   teglalap.ker:=2*(teglalap.a_oldal+teglalap.b_oldal);
   writeln('A téglalap területe: ', teglalap.ter:6:2);
   writeln('A téglalap kerülete: ', teglalap.ker:6:2);
   readln;
end.
```

*A program futásának eredménye:*

```
Téglalap a oldala: 4
Téglalap b oldala: 5
A téglalap területe:  20.00
A téglalap kerülete:  18.0
```

Készítsünk programot, amely beolvassa, és *pont* típusú rekordban tárolja két síkbeli pont koordinátáit! Számítsuk ki a két pont távolságát! (*ponttav*)

```
program ponttav;
type
     pont = record
               x, y: integer;
            end;
var
     p1, p2: pont;
     tavolsag: real;

begin
  writeln('1. koordinátapont ');
  write('   x: '); readln(p1.x);
  write('   y: '); readln(p1.y);
```

```
  writeln('2. koordinátapont ');
  write('   x: '); readln(p2.x);
  write('   y: '); readln(p2.y);
  tavolsag:= sqrt(sqr(p1.x-p2.x)+ sqr(p1.y-p2.y));
  writeln('Távolság: ',tavolsag:8:4);
  readln;
end.
```

*A program futásának eredménye:*

```
1. koordinátapont
   x: 1
   y: 1
2. koordinátapont
   x: 2
   y: 2
Távolság:   1.4142
```

Írjunk programot, amely beolvassa a megadott elemszámú tömb elemeit, azaz az *adat* típusú rekordok *x* és *y* mezőit! Számítsuk ki elemenként a beolvasott két adat átlagát, valamint az átlagok átlagát! (*atlagok*)

```
program atlagok;
type adat = record
               x, y: integer;
               atlag: real;
            end;
     tomb = array[1..20] of adat;
var
  t : tomb;
  i, db : integer;
  atlagok_atlaga: real = 0;
begin
  repeat
     write('Adatok száma: ');
     readln(db);
  until (db>0) and (db<=20);
  for i:=1 to db do
  begin
    write('1. adat: '); readln(t[i].x);
    write('2. adat: '); readln(t[i].y);
    t[i].atlag:= (t[i].x + t[i].y)/2;
    writeln('átlag: ',t[i].atlag:6:2);
    writeln;
  end;
  for i:=1 to db do
  begin
    atlagok_atlaga := atlagok_atlaga+t[i].atlag;
  end;
  atlagok_atlaga:= atlagok_atlaga/db;
  writeln('Átlagok átlaga: ',atlagok_atlaga:6:2);
  readln;
end.
```

*A program futásának eredménye:*

```
Adatok száma: 3
1. adat: 1
2. adat: 2
átlag:   1.50

1. adat: 3
2. adat: 4
átlag:   3.50

1. adat: 5
2. adat: 6
átlag:   5.50

Átlagok átlaga:   3.50
```

Készítsünk programot, amely téglalap, háromszög és kör adatait variálható rekordban tárolja! Készítsünk ebből a rekordból ötelemű tömböt, és töltsük fel véletlenszerű adatokkal! Jelenítsük meg az egyes alakzatok típusát, kerületét és területét! (*geometria*)

```
program geometria;
type
  alakzat = packed record
              terulet, kerulet : real;
              case tipus : byte of
                 1: (ta, tb : real);        // téglalap
                 2: (ha ,hb, hc : real);    // háromszög
                 3: (r : real);             // kör
            end;
const
  alakzatnev : array[1..3] of string =('téglalap', 'háromszög',
                                       'kör      ');
var
  alakzatok : array[1..5] of alakzat;
  i, n : integer;
  s : real;

begin
   // feltöltés véletlen adatokkal
   randomize;
   for i:=1 to 5 do
     begin
       n:=random(3) + 1;  // 1,2 vagy 3
       with alakzatok[i] do
         case n of
           1 : begin
                 tipus := 1;
                 ta := random(23)+1;
                 tb := random(12)+1;
                 terulet := ta*tb;
                 kerulet := 2*(ta+tb);
               end;
```

```
            2 : begin
                  tipus := 2;
                  // a háromszög szerkeszthetőségének biztosítása
                  repeat
                     ha := random(12)+1;
                     hb := random(23)+1;
                     hc := random(23)+1;
                  until (ha+hb>hc) and (ha+hc>hb) and (hb+hc>ha);
                  kerulet := (ha+hb+hc);
                  s := kerulet/2;
                  terulet := sqrt(s*(s-ha)*(s-hb)*(s-hc));
                end;
            3 : begin
                  tipus := 3;
                  r := random(12)+1;
                  terulet := sqr(r)*pi;
                  kerulet := 2*r*pi;
                end;
          end;
      end;

    // az alakzatok kerületének és területének megjelenítése
    for i:=1 to 5 do
     with alakzatok[i] do
      begin
        write(alakzatnev[tipus],#9);
        write('kerület : ', kerulet:8:3,#9);
        writeln('terület : ', terulet:8:3);
      end;
    readln;
end.
```

*A program futásainak eredménye:*

```
téglalap        kerület :   30.000      terület :   44.000
háromszög       kerület :   33.000      terület :   13.395
kör             kerület :    6.283      terület :    3.142
téglalap        kerület :   46.000      terület :  132.000
téglalap        kerület :   20.000      terület :    9.000

háromszög       kerület :   31.000      terület :   33.667
téglalap        kerület :   62.000      terület :  220.000
kör             kerület :   75.398      terület :  452.389
téglalap        kerület :    8.000      terület :    3.000
háromszög       kerület :   26.000      terület :   21.633
```

## 7.3 Halmaztípusok

Bizonyos feladatok megoldását elegánssá és hatékonnyá teszik a halmazok használata. A matematikai definíciónak megfelelően, a Pascal halmazok is bizonyos tulajdonságú elemek összessége. A Free Pascal a halmazok alkalmazását a **set** (halmaz) típussal támogatja. Egy halmaz legfeljebb 256 elemet tartalmazhat, és az alaptípusa – vagyis az elemeinek típusa – csak sorszámozott típus lehet.

A halmazok deklarációjának általános formája:

```
type
   halmaztípus = set of alaptípus;
```

A halmaz elemeit halmazváltozókban tároljuk, ahol minden lehetséges elem legfeljebb egyszer szerepelhet. A halmaz elemei nem a deklarációban, hanem az értékadás során kerülnek a halmazba. Az elemeket szögletes zárójelek között adjuk meg résztartománnyal, illetve az elemek felsorolásával.

```
type  // halmaztípusok
  tantargy = set of ( magyar, nyelvtan, matematika, fizika, angol,
                      biologia, foldrajz );
  abc = set of 'a'..'z';

var   // halmazváltozók
  targyak, szerda : tantargy;
  szunnap     : tantargy;
  szamok      : set of 0..123;
  betuk       : abc;
  hexajegyek : set of char;

begin // értékadás a halmazváltozóknak
  targyak := [magyar..foldrajz];    // minden tantárgy
  szunnap := [];                    // üres halmaz
  szerda  := [angol, matematika, foldrajz];
  betuk := [];              // az üres halmaz
  betuk := ['a'..'z'];      // a teljes halmaz
  betuk := ['n','a'];       // az 'n' és 'a' betűket tartalmazó halmaz
  hexajegyek := ['0'..'9', 'A'..'F'];
end.
```

A Free Pascal a 32-nél kisebb számosságú halmazok tárolására 32 bitet használ (***longint*** típus), míg a nagyobb számosságú halmazokat 32 bájton tárolja.

### 7.3.1 Konstans halmazok

A Pascal nyelv lehetővé teszi halmazkonstansok használatát a programban. A fenti példában az értékadás jobb oldalán ilyen halmazok szerepeltek. A halmazkonstansokat inicializálásra, és kifejezésekben egyaránt felhasználhatjuk.

A kezdőértékadás típusos halmazkonstansok és halmazváltozók esetén egyaránt alkalmazhatjuk:

```
const
  index : set of byte = [12..23];
  KisABC : set of char = ['a'..'z'];
  NagyABC : set of char = ['A'..'Z'];
  maganhangzok : set of char = ['a','e','i','o','u'];
var
  lottoszamok : set of 1..90 = [5, 2, 3, 1, 4];
```

### 7.3.2 Műveletek a halmazokkal

A halmazműveleteket két csoportra oszthatjuk. Az első csoport az ún. Boole-algebra műveleteit tartalmazza, melyek segítségével meghatározhatjuk a halmazok közös részét ($\cap$), unióját ($\cup$) és különbségét (\).

Halmazok esetén is használhatjuk a már megismert relációs operátorok többségét (kibővítve az elemvizsgálat operátorával). Természetesen az összehasonlító műveletek értelmezése valamelyest eltér az eddigiektől.

#### *7.3.2.1 Halmazok egyesítése, metszete és különbsége*

A kétoperandusú halmazműveletek operandusai csak kompatibilis halmazok lehetnek. A műveletek eredménye szintén halmaz lesz.

| *operátor* | *művelet* | *precedncia* |
|---|---|---|
| * | halmazok metszete (közös része) | elsődleges |
| + | halmazok egyesítése (uniója) | másodlagos |
| - | halmazok különbsége | másodlagos |

Az egyes műveletek bemutatása során az *A* és *B* halmazokra hivatkozunk. Definiáljuk mindkét halmazt az alábbi programrészletnek megfelelően!

```
type
  halmaz = set of byte;

const
  A : halmaz = [1, 2, 3, 4];
  B : halmaz = [1, 3, 5, 6];
```

| *Művelet* | *Leírás* | *Eredményhalmaz* |
|---|---|---|
| Halmazok metszete:<br>A*B | Két halmaz metszete (közös része) olyan halmaz, amelynek elemei mindkét halmaznak elemei. | `[1,3]` |
| Halmazok egyesítése:<br>A+B | Két halmaz egyesítése olyan halmaz, amely mindkét operandushalmaz elemeit tartalmazza. Az eredményhalmazban a közös elemek csak egyszer szerepelnek. | `[1,2,3,4,5,6]` |
| Halmazok különbsége:<br>A-B<br>B-A | Két halmaz különbsége olyan halmaz, amelynek minden eleme benne van az első halmazban, de nem eleme a második halmaznak. | `[2,4]`<br>`[5,6]` |

#### *7.3.2.2 Halmazkezelő eljárások*

Az ***include()*** és az ***exclude()*** eljárások segítségével a halmazhoz hozzáadható, illetve a halmazból kivehető egy elem. Ezek az eljárások a hagyományos egyesítés és különb-

ség műveleteknél sokkal hatékonyabban működnek. Az eljárások segítségével a halmazok elemeire vonatkozó utasítások az alábbi formában is megadhatók:

```
include(halmaz, elem);          halmaz := halmaz + [elem]
exclude(halmaz, elem);          halmaz := halmaz - [elem]
```

#### *7.3.2.3 Halmazokra vonatkozó vizsgálatok*

Két halmazról megállapíthatjuk, hogy egyenlők-e (azonos elemeket tartalmaznak-e) vagy sem. Halmazok esetén különböző tartalmazás-vizsgálatokat is végezhetünk. Megvizsgálhatjuk, hogy egy adott érték eleme-e a halmaznak ($\in$), illetve azt, hogy egy halmaz minden eleme megtalálható-e egy másik halmazban ($\subset$). Ha az utóbbi vizsgálat igaz eredményt ad, akkor azt mondjuk, hogy az egyik halmaz a másik részhalmaza.

| *operátor* | *művelet* |
|---|---|
| = | azonos, |
| <> | nem azonos, |
| <= | részhalmaz - a műveleti jel bal oldalán álló halmaz a jobb oldalon álló halmaz részhalmaza, |
| >= | részhalmaz - a jobb oldalon álló halmaz a bal oldalon álló halmaz részhalmaza, |
| **in** | az eleme viszony vizsgálata - ha (*adat* **in** *halmaz*) kifejezés értéke igaz (**true**), akkor az *adat* benne van a *halmaz*ban. |

### 7.3.3 Halmazkezelési példaprogramok

Érdemes megjegyezni, hogy a halmazműveletek általában gyorsabbak, és a forráskódot is tömörebbé teszik, mint a hagyományos megoldások. Nézzünk néhány gyakran használt megoldást!

```
// csak a megfelelő válasz megadása után fut tovább a program
var
  valasz : char;
begin
  repeat
    write('Igen/Nem: '); readln(valasz);
  until valasz in ['I','i','N','n'];
end.

// várakozás 1 és 23 közé eső (zárt intervallumban) szám bevitelére
var
  elemszam : integer;
begin
  repeat
    write('Elememszám 1-23: '); readln(elemszam);
  until elemszam in [1..23];
end.
```

Írjunk programot, amely megállapítja egy halmaz számosságát (elemeinek számát), és megjeleníti a halmaz elemeit! (*halmazkiiras*)

```
program halmazkiiras;
type
  nagybetuk = 'A'..'Z';
var
  halmaz : set of nagybetuk = ['X','Z', 'A'..'F', 'K', 'P'..'R'];
  szamossag : integer = 0;
  ch : nagybetuk;
begin
  for ch:=low(nagybetuk) to high(nagybetuk) do
   if ch in halmaz then
      begin
         inc(szamossag);
         write('''',ch,'''',#32);
      end;
   writeln;  // 'A' 'B' 'C' 'D' 'E' 'F' 'K' 'P' 'Q' 'R' 'X' 'Z'
   writeln('A halmaz számossága: ', szamossag);   // 12
  readln;
end.
```

*A program futásának eredménye:*

```
'A' 'B' 'C' 'D' 'E' 'F' 'K' 'P' 'Q' 'R' 'X' 'Z'
A halmaz számossága: 12
```

Készítsünk programot, amely egy sztringtömb elemeit minden futásnál más és más sorrendben jeleníti meg! (*sorsolas*)

```
program sorsolas;

const
  maxn = 7;

type
  indexek = 1..maxn;

const
  nevek : array[indexek] of string = ('Anna', 'Natália',
                                      'Léna', 'Júlia',
                                      'Adrienn', 'Iván',
                                      'Mária');
var
  indexhalmaz : set of indexek;
  i, n : integer;

begin
  // az indexek halmazának feltöltése
  for i:=low(indexek) to high(indexek) do
    include(indexhalmaz, i);

  // sorsolás
  randomize;
```

```
  for i:=low(indexek) to high(indexek) do
    begin
       repeat
          n:=random(maxn)+1;
       until n in indexhalmaz;
       writeln(nevek[n]);
       exclude(indexhalmaz, n);
    end;
  readln;
end.
```

*A program futásainak eredménye:*

```
Júlia
Anna
Mária
Léna
Iván
Natália
Adrienn

Adrienn
Mária
Iván
Natália
Léna
Anna
Júlia
```

Pascal program segítségével állapítsuk meg két beolvasott szöveg közös betűit, és jelenítsük meg az eredményt! (*kozosbetuk*)

```
program kozosbetuk;
var
  szoveg1, szoveg2 : string;
  betuk1, betuk2, kozos : set of char;
  i : integer;

begin
  write('Az első szöveg   : '); readln(szoveg1);
  write('A második szöveg : '); readln(szoveg2);

  // Az angol ABC betűinek átvitele a halmazokba
  betuk1:=[];
  for i:=1 to length(szoveg1) do
    if szoveg1[i] in ['A'..'Z', 'a'..'z'] then
       betuk1 := betuk1 + [szoveg1[i]];

  betuk2:=[];
  for i:=1 to length(szoveg2) do
    if szoveg2[i] in ['A'..'Z', 'a'..'z'] then
       betuk2 := betuk2 + [szoveg2[i]];

  // A közös betűket a metszethalmaz tartalmazza
  kozos := betuk1 * betuk2;
```

```
  // Az eredményhalmaz elemeinek megjelenítése
  write('Közös betűk      : ');
  for i:=0 to 255 do
    if chr(i) in kozos then write(chr(i):2);
  writeln;
  readln;
end.
```

*A program futásának eredménye:*

```
Az első szöveg    : Nem kell mar varnod a hajnal jottet
A második szöveg  : Nem kell mar felned a naplementet
Közös betűk       :  N a d e k l m n r t
```

Az Eratosztenész szitája módszer segítségével keressük meg a prímszámokat 1 és 255 között! (eratosztenesz)

```
program Eratosztenesz;
var
  i, k : integer;
  szita : set of byte;
  n, prim : byte;

begin
  repeat
    write('Meddig keressük a prímszámokat?: ');
    readln(n);
  until n in [2..255];

  // 2 és n közötti számokkal feltöltjük a szitát
  for i:=2 to n do
    include(szita, i);

  // A nem prímszámok kiszűrése
  prim := 2;
  while sqr(prim)<=n do
    begin
       // az összes k*prim kivétele szitából, k>1
       k := 2*prim;
       while k<=n do
         begin
           exclude(szita, k);
           inc(k, prim);
         end;
      // a következő legkisebb prímszám keresése
      repeat inc(prim) until prim in szita;
    end;

  for i:=2 to n do
    if i in szita then write(i:4);
  writeln;
  readln;
end.
```

*A program futásainak eredménye:*

```
Meddig keressük a prímszámokat?: 35
   2   3   5   7  11  13  17  19  23  29  31

Meddig keressük a prímszámokat?: 60
   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59
```

Összehasonlításképpen, valósítsuk meg az Eratosztenész szitája módszert halmaz használata nélkül! (*primektomb*)

```
program primektomb;
var
  i, k : integer;
  szita : array[byte] of boolean;
  n, prim : byte;

begin
  repeat
    write('Meddig keressük a prímszámokat?: ');
    readln(n);
  until n in [2..255];

  // 2 és n közötti számokkal feltöltjük a szitát
  for i:=2 to n do
    szita[i]:=true;

  // A nem prímszámok kiszűrése
  prim := 2;
  while sqr(prim)<=n do
    begin
       // az összes k*prim kivétele szitából, k>1
       k := 2*prim;
       while k<=n do
         begin
           szita[k]:=false;
           inc(k, prim);
         end;
      // a következő legkisebb prímszám keresése
      repeat inc(prim) until szita[prim];
    end;

  for i:=2 to n do
    if szita[i] then write(i:4);
  writeln;
  readln;
end.
```

## 7.4 Állománytípusok

Az állomány (fájl, **file**) azonos típusú elemekből felépülő adatszerkezet, amely nagymennyiségű adat tárolását teszi lehetővé a háttértárolón. A fájl komponensei egyaránt lehetnek egyszerű és strukturált típusú adatok, de leggyakrabban a ***record*** típust használjuk. A típusos fájl változóját az alábbi formában deklaráljuk:

```
var
   típusos_fájl : file of elemtípus;
```

Elérhetünk állományokat úgy is, hogy nem adjuk meg a típust – ezek a típus nélküli állományok. (A típus nélküli fájl, **byte** típusú állományként is felfogható).

```
var
   típus_nélküli_fájl : file;
```

Gyakran használunk szövegsorokat tartalmazó szöveges állományokat:

```
var
   szövegfájl1 : text;
   szövegfájl2 : textfile;
```

Az állományok kezelésével részletesebben külön fejezetben foglalkozunk.

## 7.5 Objektumtípusok

A Free Pascal több módon is támogatja az objektum-orientált programozást. Ennek egyik eszköze az **object** típus, amely már a ***Turbo Pascal*** későbbi változataiban is elérhető volt. Az objektumtípus sokban emlékeztet a rekordtípusra, azonban az adatmezőkön túlmenően alprogramokat (metódusokat) is tartalmazhat. További különbség az objektumelemek elérhetőségének szabályozása, illetve az egymásba építés helyett, az öröklés alkalmazása. Az objektum-definíció általános formája:

```
type
   objektumtípus = ⟨packed⟩ object⟨(ősobjektum)⟩
    ⟨public | private | protected⟩
      adatmező-deklarációk;
      metódusfejlécek;
   end;

metódusdefiníciók;
```

Az objektumtípus használatát részletekbe menően a 11. fejezetben ismertetjük.

## 7.6 Osztály- és interfésztípusok

A Free Pascal ***ObjFPC*** és ***Delphi*** módjaiban az objektum-orientált programozás egy másik megközelítését is támogatja a **class** típussal. A **class** típus adatok (*field*), alprogramok (*method*) és tulajdonságok (*property*) együttesét definiálja. Az osztálytí-

pus alapján elkészített változókat (példányokat) objektumnak nevezzük. Alapvető különbség az **object** típushoz képest, hogy a **class** típus változói mindig dinamikusan jönnek létre. A **class** típus teszi lehetővé a Free Pascal nyelv grafikus felületű alkalmazások fejlesztésére történő alkalmazását. Az osztálytípus deklarálásának általános formája:

```
type
   osztálytípus = ⟨packed⟩ class ⟨ (ősosztály ⟨,interfészlista⟩) ⟩
    ⟨public | published | private | protected⟩
      adatmező-deklarációk;
      metódusfejlécek;
      tulajdonság-definíciók;
   end;

metódusdefiníciók;
```

A Free Pascal **object** és **class** típusaival megvalósított egyszeres öröklődés lehetetlenné tenné megosztott objektumok használatát, mint például a ***COM*** (*Component Object Model*), a ***CORBA*** (*Common Object Request Broker Architecture*). Ezért szükség van egy olyan megoldásra, amely viszonylag egyszerű támogatást nyújt a fenti technológiák elérésére. Free Pascalban ennek az eszköznek a neve **interface**.

Az **interface** sokban emlékeztet az osztályra, azonban a deklarált metódusait nem definiáljuk, és nem készítünk belőle objektumpéldányokat. Az interfészt az osztályok készítése során használhatjuk, ahol meg kell valósítanunk az **interface** metódusait. Az interfésztípus alkalmazása elérhetővé teszi a többszörös öröklés előnyeit, a Pascal nyelv alapvető átalakítása nélkül. Az interfész általános deklarációja:

```
type
   interfésztípus = interface ⟨ (interfészlista) ⟩
      [' GUID ']
      metódusfejlécek;
      tulajdonság-definíciók;
   end;
```

A *GUID* (*Globally Unique Identifier*) egy 128-bites egyedi egész érték, amely az interfész operációs rendszer alatti azonosítását szolgálja.

A **class** és az **interface** típusok használatával kapcsolatos ismereteket, könyvünk 12. fejezete részletezi.

# 8. Eljárások és függvények

A Pascal program szerkezete áttekinthetőbb, rövidebb és strukturáltabb, ha a logikailag összefüggő utasításokat alprogramokba foglaljuk.

A Pascal nyelvben kétfajta alprogram használható: az eljárás (**procedure**) és a függvény (**function**), melyek a hívás módjában különböznek egymástól.

Az alprogram utasításainak végrehajtásához bemenő adatokra van szükség, melyeket a paraméterein keresztül kap meg. Ugyancsak pontosan meg kell adni, hogy hol keletkezzen az alprogram működésének eredménye. Az alprogram definíciójában a paramétereket formális paramétereknek, míg az eljárást és függvényt hívó utasításban szereplő paramétereket aktuális (hívási) paramétereknek nevezzük. A hívási paramétereket argumentumoknak is szokás nevezni.

## 8.1 Eljárások és függvények paraméterei

A Pascal nyelvben a paramétereket hagyományosan két csoportba sorolhatjuk: érték- és változóparaméterek. A Free Pascal minden módban a konstansparaméterekkel, az *ObjFpc* (és *Delphi*) módban pedig a kimenő paraméterekkel bővíti a palettát.

### 8.1.1 Érték szerinti átadott, bemenő paraméterek

Az érték szerinti paraméterátadásnál – az aktuális paraméter felülírás elleni védelme érdekében – a bejövő értékek átmásolódnak egy munkaterületre, és ezeket a másolatokat használja az alprogram. Ha egy bemenő paraméter értékét az eljárásban megváltoztatjuk, akkor csak a másolat módosul, így a híváskor megadott aktuális paraméter értéke változatlan marad.

Az értékparaméterek helyén a híváskor megadhatunk konstanst, változót, illetve tetszőleges kifejezést, melyek típusa kompatibilis a formális paraméter típusával.

A paraméterlista megadásának szabályai:

- A paramétereket mindig kerek zárójelbe kell tenni.
- Az azonos típusú paramétereket vesszővel kell elválasztani, az utolsó paraméter után kettőspontot téve a paraméter típusát kell megadni.
- A különböző típusú paramétereket pontosvesszővel választjuk el egymástól.

```
(x, y: real; db: integer; adat: string)
```

### 8.1.2 Cím szerinti átadott be-/kimenő paraméterek

A cím szerinti paraméterátadás során az alprogram átveszi a hívási paraméter címét, melynek segítségével a formális paraméter valójában az aktuális paraméter másik (alias) neveként jelenik meg. Ennek következtében az alprogramon belül elvégzett módosítások közvetlenül a híváskor megadott változón fejtik ki hatásukat. A változóparaméterben az alprogram számára bemenő adatokat adhatunk meg, illetve az eredményeket is megkaphatjuk. Mivel a cím szerinti paraméterátadás során csak a változó címe másolódik, sokkal gyorsabb és helytakarékosabb az alprogram hívása, mint értékparaméterek használata esetén.

A paraméterlistán úgy különböztetjük meg az érték szerinti formális paramétereket a cím szerinti formális paraméterektől, hogy a paraméterek előtt megadjuk a **var** kulcsszót.

A cím szerint átadott hívási (aktuális) paraméter csak a formális paraméterrel megegyező típusú változó lehet. Ha egy alprogram különböző típusú változóparaméterrel rendelkezik, akkor mindegyik paraméter elé ki kell tenni a **var** kulcsszót.

```
(a_oldal, b_oldal: real; var terulet, kerulet: real;
 var valasz: integer)
```

### 8.1.3 Konstans-, bemenő paraméterek

A cím szerinti paraméterátadás hatékonyságát és az érték szerinti paraméterátadás biztonságát ötvözi a konstansparaméter. Az értékparaméterekkel ellentétben azonban, a konstansparaméter értéke az alprogramon belül nem módosítható.

A paraméterlistán a konstans, formális paraméterek előtt a **const** kulcsszót használjuk.

Híváskor, a konstansparaméterek helyén – az értékparaméterekhez hasonlóan – konstanst, változót, illetve tetszőleges kifejezést egyaránt megadhatunk, melyek típusa kompatibilis a formális paraméter típusával.

```
(const a_oldal, b_oldal: real; var terulet, kerulet: real)
```

### 8.1.4 Csak Kimenő paraméterek

Az *ObjFpc* (és *Delphi*) módban a cím szerinti paraméterátadás speciális alkalmazását jelentik a kimenő paraméterek. Ekkor a paramétert csak az eredmény visszaadására használjuk, bemenő értékét nem szabad felhasználni.

A paraméterlistán a kimenő, formális paraméterek előtt az **out** kulcsszót használjuk.

Híváskor, a kimenő paraméterek helyén – a változóparaméterekhez hasonlóan – csak a formális paraméterrel megegyező típusú változót adhatunk meg.

```
(const a_oldal, b_oldal: real; out terulet, kerulet: real)
```

## 8.2 Függvények

A függvények különlegessége abban áll, hogy a nevükön keresztül egy értéket képesek visszaadni. A függvények általános leírása:

```
function függvénynév (formális paraméterek listája): függvénytípus;
 // lokális deklarációs rész
 // label, const, type, var, procedure, function deklarációk
begin
    // a függvény törzse - végrehajtható utasítások
    függvénynév := kifejezés;
end;
```

*ObjFpc* (és *Delphi*) módban a függvény visszatérési értékét az előre definiált **result** változó segítségével is megadhatjuk:

```
result := kifejezés;
```

Néhány megjegyzés a függvények készítésével kapcsolatban:

- A függvény azonosítóját (***függvénynév***) a Pascal nyelv névképzési szabályai szerint kell kialakítani.
- A függvény paraméterei általában értékparaméterek, mivel ebben adjuk át a bemenő adatokat. A függvények esetén változóparamétert legtöbbször csak hibajelzés céljából használunk.
- A paraméterlistát lezáró kerek zárójel és egy kettőspont után adjuk meg a függvény típusát, ami határozza, hogy a függvény milyen típusú értéket ad vissza. Free Pascalban a függvények tetszőleges (egyszerű és összetett) típusú adattal visszatérhetnek.
- A függvény lokális deklarációit a függvény fejléce alatt adjuk meg, ezeket csak a függvényen belül használhatjuk.
- A függvény törzse **begin end**; között tartalmazza azokat az utasításokat, amelyek az algoritmusnak megfelelően meghatározzák az eredményt, azaz a függvény egyetlen visszatérési értékét.
- A függvény nevét (vagy a ***result*** változót) az értékadó utasítás bal oldalán szerepeltetve határozzuk meg a visszatérési értéket. A függvény törzsén kívül a függvény azonosítója önmagában nem szerepelhet az értékadó utasítás bal oldalán (bizonyos kifejezésekben azonban igen).

- Free Pascalban a függvény neve – a zárójelek és argumentumlista nélkül – az értékadás jobb oldalán is állhat – mintegy a függvénytípussal egyező típusú változó.
- A függvény törzsének tetszőleges pontján kiléphetünk az alprogramból az **exit** utasítás használatával. Az utasítás után zárójelben megadhatjuk a függvényértéket. Az **exit** paraméter nélküli formáját használva, a hívás előtt magunknak kell beállítani a függvény értékét a fentiekben ismertetett módon.

### *A függvények hívása:*

A függvényhívást általában az értékadás jobb oldalán, illetve kifejezésekben szerepeltetjük. Híváskor a függvény nevét zárójelben követik az aktuális paraméterek (argumentumok), melyek számának egyeznie kell a formális paraméterek számával. Változó- és kimenő paraméterek esetén a típusegyezőség is követelmény, azonban érték- és konstansparaméterek használatakor elegendő, ha az argumentum típusa átalakítható a formális paraméter típusává.

```
változó := függvénynév(argumentumlista);
```

Amennyiben nincs szükségünk a függvény értékére, a függvényhívás önállóan is szerepelhet az utasításban.

A paraméter nélküli függvények hívásakor a függvény nevét önállóan, illetve üres zárójelpárral kiegészítve egyaránt használhatjuk:

```
változó := függvénynév;
változó := függvénynév();
```

Free Pascalban a függvényen belül csak a második esetben valósít meg (rekurzív) hívást a fordító:

*276* értékkel visszatérő függvény:

```
function fv1() : integer;
begin
  fv1:=12;
  fv1:=fv1*23;
end;
```

Önmagát végtelenszer hívó függvény:

```
function fv1() : integer;
begin
  fv1:=12;
  fv1:=fv1()*23;
end;
```

Megjegyezzük, hogy paraméter nélküli esetben a függvény fejlécében is elhagyhatjuk a zárójeleket.

```
function fv1 : integer;
begin
  fv1:=12;
  fv1:=fv1*23;
end;
```

Azonban sokkal áttekinthetőbb programhoz jutunk, ha a függvény fejlécében, illetve a függvény hívásakor egyaránt megadjuk az üres zárójelpárt.

Tervezzünk *hatvany* függvényt, amely a valós alapot pozitív, vagy negatív egész kitevőre emeli! (*hatvanyoz*)

*A tervezés menete:*
A függvény neve: *hatvany*
Két értékparaméterrel rendelkezik:

| *Paraméter* | *leírás* | *deklaráció* |
|---|---|---|
| *a* | a valós alap | `a:real` |
| *k* | az egész kitevő | `k:integer` |

Az eredmény szintén valós szám. A függvény fejléce:

```
function hatvany(a: real; k: integer): real;
```

*A függvény lokális deklarációi:*

*i* a ciklus egész típusú változója,
*h* valós változó, amely a hatványt tartalmazza.

```
var
   h: real;
   i: integer;
```

*A függvény algoritmusa:*
A hatványt ismételt szorzással állítjuk elő. Az alapot pozitív hatványra emeljük úgy, hogy a ciklus 1-től a kitevő abszolút értékéig fut. Majd megvizsgáljuk a kitevő előjelét, ha a kitevő negatív a hatvány reciprokát vesszük.

*A függvény törzse:*

```
begin
   h:= 1;  // alapállapot
   // a ciklus 1-től a kitevő abszolút értékéig fut
   for i:=1 to abs(k) do
      h:= h*a;       // az alappal szorzunk
   if k >= 0 then    // a kitevő előjelének vizsgálata
     hatvany:= h     // a függvényérték nem negatív esetben
   else
     hatvany:= 1/h; // a függvényérték negatív kitevő esetén
end;
```

*A függvény használatát bemutató program:*

```
program hatvanyoz;
{$Mode TP}
var
   alap, hat: real;
   kitevo: integer;
```

```
function hatvany(a: real; k: integer): real;
var
   h: real;
   i: integer;
begin
   h:= 1;
   for i:=1 to abs(k) do
     h:= h*a;
   if k >= 0 then
     hatvany:= h
   else
     hatvany:= 1/h;
end;

begin
   write('alap: '); readln(alap);
   write('kitevő: '); readln(kitevo);
   hat:= hatvany(alap,kitevo); { a függvény hívása jobb oldalon}
   writeln('hatvány értéke: ',hat:8:4);
   { a függvény hívása writeln utasításban }
   writeln('hatvány értéke: ',hatvany(2,3):8:4);
   writeln('hatvány értéke: ',hatvany(2,-3):8:4);
   readln;
end.
```

*A program futásának eredménye*

```
alap: 4
kitevő: 2
hatvány értéke:  16.0000
hatvány értéke:   8.0000
hatvány értéke:   0.1250
```

Írjuk át a *hatvany* függvényt a Free Pascal eszköztárának felhasználásával!

A fenti megoldásban vigyáztunk arra, hogy a ***Turbo Pascal*** rendszernek megfelelő ***TP*** módban is lefordítható legyen. A függvény törzse egyszerűbben megvalósítható a *h* segédváltozó bevezetése nélkül:

```
function hatvany(a: real; k: integer): real;
var
   i: integer;
begin
   hatvany:= 1;
   for i:=1 to abs(k) do
     hatvany:= hatvany*a;
   if k < 0 then
     hatvany:= 1/hatvany;
end;
```

A megoldásunk az alapértelmezett ***Fpc*** valamint az ***ObjFpc*** módokban jól működik, azonban ***Delphi*** módban fordítási hibákat kapunk. A problémát a **result** változó bevezetésével kiküszöbölhetjük:

```
function hatvany(a: real; k: integer): real;
var
   i: integer;
begin
   result:= 1;
   for i:=1 to abs(k) do
     result:= result*a;
   if k < 0 then
     result:= 1/result;
end;
```

## 8.3 Eljárások

Az eljárások, eltérően a függvényektől, a nevükön keresztül nem adnak vissza értéket, így hívásuk önálló Pascal utasításként jelenik meg. Míg a függvényeket – az esetek többségében – egyetlen érték meghatározására használjuk, addig az eljárások alkalmazásának célja valamilyen tevékenység elvégzése. Az eljárások felépítésének általános formája:

```
procedure eljárásnév (formális paraméterek listája);
 // lokális deklarációs rész
 // label, const, type, var, procedure, function deklarációk
begin
    // az eljárás törzse - végrehajtható utasítások
end;
```

Az eljárásokra is érvényesek a függvényeknél elmondottak többsége, így ezeket itt nem ismételjük meg. Néhány megjegyzést azonban mégis teszünk az eljárások készítésével kapcsolatban:

- Az eljárás azonosítóját (***eljárásnév***) szintén a Pascal nyelv névképzési szabályai szerint kell kialakítani.
- Az eljárások bemenő adatait általában értékparaméterekben adjuk meg, míg a működésének eredményét változóparamétereken keresztül kapjuk meg.
- Az eljárás lokális deklarációit az eljárás fejléce alatt adjuk meg, azonban ezeket csak az eljáráson belül használhatjuk.
- Az eljárás törzse **begin end**; között tartalmazza azokat az utasításokat, amelyek az algoritmusnak megfelelően elvégzik a szükséges tevékenységet.
- Az eljárás törzsébe helyezett **exit** utasítás segítségével, bármikor visszatérhetünk az alprogramból.

### *Az eljárások hívása:*

A függvényhívással szemben az eljáráshívás önálló utasítása a Pascal nyelvnek. Híváskor az eljárás nevét zárójelben követik az aktuális paraméterek (argumentumok), melyek használatára vonatkozó szabályok megegyeznek a függvényhívásnál elmondottakkal.

```
eljárásnév(argumentumlista);
```

A paraméter nélküli eljárások hívásakor az eljárás nevét önállóan, illetve üres zárójelpárral kiegészítve egyaránt használhatjuk, azonban itt is a második forma alkalmazása a javasolt:

```
eljárásnév;
eljárásnév();
```

Tervezzünk eljárásokat, amelyek jól használhatók téglalapok oldalhosszainak beolvasása, illetve a terület és a kerület számítása során! (*alprg*)

Induljunk ki egyetlen főprogramot tartalmazó megoldásból, mely alapján elkülöníthetjük az elvégzendő tevékenységeket!

```
program MintaPrg;
var
   a_oldal, b_oldal, terulet, kerulet: real;
begin
  write('Téglalap a oldala: '); readln(a_oldal);
  write('Téglalap b oldala: '); readln(b_oldal);
  terulet:= a_oldal * b_oldal;
  kerulet:= 2*(a_oldal + b_oldal);
  writeln('Téglalap területe: ',terulet:6:2);
  writeln('Téglalap kerülete: ',kerulet:6:2);
  readln;
end.
```

*A program futásának eredménye:*

```
Téglalap a oldala: 2
Téglalap b oldala: 3
Téglalap területe:   6.00
Téglalap kerülete:  10.00
```

A feladat megoldásához szükséges lépéseket három csoportba sorolhatjuk, melyek megvalósításához eljárásokat készítünk:

- a téglalap két oldalának beolvasása - *AdatokatOlvas*,
- a kerület és a terület meghatározása - *TerKerSzamol*,
- az eredmények megjelenítése - *EredménytKiir*.

Az *AdatokatOlvas* eljárás bekéri a téglalap két oldalának hosszát, majd az értékeket két cím szerint átadott paraméteren keresztül visszaadja.

| *paraméter* | *leírás* | *deklaráció* |
|---|---|---|
| *a* | a téglalap egyik oldala | `var a :real` |
| *b* | a téglalap másik oldala | `var b :real` |

Az eljárás fejléce:

```
procedure AdatokatOlvas(var a, b: real);
```

Ennek az eljárásnak nincsenek lokális változói, mivel az *a*, *b* változóparaméterekbe olvassuk be az adatokat. Természetesen a beolvasás előtte a ***write()*** eljárással rákérdezünk az adatra:

```
procedure AdatokatOlvas(var a, b: real);
begin
  write('Téglalap a oldala: '); readln(a);
  write('Téglalap b oldala: '); readln(b);
end;
```

A *TerKerSzamol* eljárás feladata a téglalap két oldalából kiszámítani a téglalap kerületét és a területét. Az oldalhosszakat értékparaméterekben kapja meg az eljárás, míg a számított adatokat változóparaméterben adja vissza:

| *paraméter* | *leírás* | *deklaráció* |
|---|---|---|
| *a* | a téglalap egyik oldala | `a :real` |
| *b* | a téglalap másik oldala | `b :real` |
| *ter* | a téglalap területe | `var ter :real` |
| *ker* | a téglalap kerülete | `var ker :real` |

```
procedure TerKerSzamol( a, b:real; var ter, ker: real);
begin
  ter:= a * b;
  ker:= 2 * (a + b);
end;
```

Az *EredménytKiir* eljárás feladata a téglalap területének és kerületének megjelenítése. Az ehhez szükséges adatokat értékparaméterekben kapja meg:

| *paraméter* | *leírás* | *deklaráció* |
|---|---|---|
| *ter* | a téglalap területe | `ter :real` |
| *ker* | a téglalap kerülete | `ker :real` |

```
procedure EredmenytKiir(ter,ker: real);
begin
  writeln('Téglalap területe: ',ter:6:2);
  writeln('Téglalap kerülete: ',ker:6:2);
  writeln;
end;
```

A feladat teljes megoldásához még el kell készítenünk a fenti eljárásokat hívó főprogramot. Legyen a program neve: *AlPrg*!

A főprogramban deklarálnunk kell két változót a téglalap oldalai számára, valamint a két változót az eredmények: a kerület és a terület tárolására. Mivel két téglalapnak szeretnénk kiszámítani a területét és a kerületét, ezért mindegyik változóból kettőt kell deklarálnunk.

*A teljes program listája:*

```
program AlPrg;
var
   a_oldal1, b_oldal1, terulet1, kerulet1,
   a_oldal2, b_oldal2, terulet2, kerulet2: real;

procedure AdatokatOlvas(var a, b: real);
begin
  write('Téglalap a oldala: '); readln(a);
  write('Téglalap b oldala: '); readln(b);
end;

procedure TerKerSzamol( a, b:real; var ter, ker: real);
begin
  ter:= a * b;
  ker:= 2*(a + b);
end;

procedure EredmenytKiir(ter,ker: real);
begin
  writeln('Téglalap területe: ',ter:6:2);
  writeln('Téglalap kerülete: ',ker:6:2);
  writeln;
end;

begin
  AdatokatOlvas(a_oldal1, b_oldal1);
  TerKerSzamol(a_oldal1, b_oldal1, terulet1, kerulet1);
  EredmenytKiir(terulet1, kerulet1);
  AdatokatOlvas(a_oldal2, b_oldal2);
  TerKerSzamol(a_oldal2, b_oldal2, terulet2, kerulet2);
  EredmenytKiir(terulet2, kerulet2);
  readln;
end.
```

*A program futásának eredménye:*

```
Téglalap a oldala: 2
Téglalap b oldala: 3
Téglalap területe:   6.00
Téglalap kerülete:  10.00

Téglalap a oldala: 4
Téglalap b oldala: 5
Téglalap területe:  20.00
Téglalap kerülete:  18.00
```

## 8.4 Globális és lokális deklarációk

A Pascal programok általában több modulból épülnek fel, melyeket külön forrásállományok tárolják. Ezek közül a főmodul tartalmazza a főprogramot, míg a többi modul az ún. **unit**, melyek használatával kapcsolatos ismereteket a 9. fejezet fogja csokorba.

A főmodul deklarációs és végrehajtható részre tagolódik. A deklarációs részben – többek között – változókat és alprogramokat hozhatunk létre. Az alprogramok saját, belső deklarációs résszel rendelkeznek, ahol változókat és további alprogramokat deklarálhatunk. A sort folytathatnánk, azonban megelégszünk két szint, a főprogram és a főprogram deklarációs részében megadott alprogramokon belül definiált változók által felvetett kérdések tisztázásával.

### 8.4.1 Az azonosítók érvényességi köre, elérhetősége

A főprogram szintjén deklarált változók globális elérhetőséggel rendelkeznek a főmodulban, ezért ezeket globális változóknak nevezzük. A globális változók a főprogramból, és a főmodul összes, a deklarációt követő alprogramjából egyaránt elérhetők. Felhívjuk a figyelmet arra, hogy rossz programozási gyakorlat, ha a paraméterek helyett globális változókat használunk az alprogramokkal való adatcserére.

Az eljárásokban és a függvényekben deklarált változók a lokális (helyi) változók, mivel ezeket csak az alprogramon belül (a belső alprogramokból, illetve a **begin**...**end**; blokkból) érhetjük el. Az elmondottak következménye, hogy különböző alprogramokban azonos névvel, eltérő típusú változókat is deklarálhatunk. Felhívjuk a figyelmet arra, hogy a formális paraméterek is a definiáló alprogram lokális változói.

Amennyiben egy magasabb szinten (például globális) definiált változóval megegyező nevű változót deklarálunk alacsonyabb szinten (például lokális), akkor a külső szinten megadott változó elérhetetlenné válik az alprogram számára.

Készítsünk programot, amelyben lekérdezhetjük a Fibonacci számsor elemeit, és adott a lehetőség a számsor újraindítására! Használjunk globális változókat! (*fibo1*)

```
program fibo1;

var
  a0, a1 : integer;   // globális változók

procedure start;      // a sorozat indítása
begin
  a0 := -1;
  a1 := 1;
end;
function kovetkezo : integer;  // a sorozat következő eleme
var a2 : integer;  // lokális változó
```

```
begin
   a2:= a0 + a1;
   a0:=a1;
   a1:=a2;
   kovetkezo:=a2;
end;

var i:integer;  // globális változó - csak a főprogramból érjük el
begin
  start();
  for i:=0 to 12 do
    writeln(i:5,'.', kovetkezo():6);
  readln;
end.
```

*A program futásának eredménye:*

```
    0      0
    1      1
    2      1
    3      2
    4      3
    5      5
    6      8
    7     13
    8     21
    9     34
   10     55
   11     89
   12    144
```

A feladat megoldása során globális változókat használtunk a két alprogram közötti kommunikációra. Általában ez nem követendő példa, azonban unitba szervezve a változókat és az alprogramokat, hatékony és jó megoldáshoz jutunk.

Oldjuk meg az előző feladatot paraméterek alkalmazásával! (*fibo2*)

```
program fibo2;

procedure start(var a0, a1:integer);
begin
  a0 := -1;
  a1 := 1;
end;

function kovetkezo(var a0, a1:integer) : integer;
var a2 : integer;
begin
   a2:= a0 + a1;
   a0:=a1;
   a1:=a2;
   kovetkezo:=a2;
end;
```

```
var
  a0, a1, i : integer;
begin
  start(a0, a1);
  for i:=0 to 12 do
    writeln(i:5,'.', kovetkezo(a0, a1):6);
  readln;
end.
```

### 8.4.2 Az azonosítók élettartama

A globális változók a program indításakor létrejönnek, és elérhetők maradnak a programból való kilépésig. Ezt a jelenséget statikus élettartamnak szokás nevezni, ellentétben a lokális változók dinamikus élettartamával. A lokális változók az alprogram hívásakor létrejönnek, és megszűnnek, amikor az kilépünk az alprogramból.

Dinamikus élettartamról beszélünk akkor is, amikor mi magunk hozzuk létre a változót, illetve szüntetjük meg. A dinamikus memóriahasználat a 10. fejezet témája.

Már a Turbo Pascalban is volt lehetőség olyan lokális változók létrehozására, melyek a hívások között is megőrzik értéküket, tehát statikus élettartammal rendelkeznek. Az ilyen változók valójában az alprogramban deklarált típusos konstansok.

Példaként tekintsük a *fibo()* függvényt, amelyet **true** értékű argumentummal aktiválva megadja a Fibonacci-sorozat soron következő elemét, míg **false** argumentum esetén újraindítja a sorozatot:

```
function fibo(kovetkezo:boolean) : integer;
const
  a0 : integer =-1;    // statikus élettartamú, lokális változók
  a1 : integer = 1;
var
  a2 : integer;        // dinamikus élettartamú, lokális változó
begin
  if kovetkezo then
    begin
      a2:= a0 + a1;
      a0:=a1;
      a1:=a2;
      fibo:=a2;
    end
  else
    begin
      a0 := -1;
      a1 := 1;
      fibo:=0;
    end;
end;
```

## 8.5 A forward (előzetes) deklaráció

Az előzetes deklarációt akkor alkalmazzuk, amikor egy függvényt vagy eljárást előbb használunk (hívunk), mint ahogy deklarálunk. Az előzetes deklarációban az alprogram fejlécét **forward** foglalt szó követi. Free Pascalban az alprogram definiálásakor a teljes fejlécet meg kell ismételni, nem lehet a paramétereket elhagyni, mivel ekkor egy másik mechanizmus, az ún. overloading (túlterhelés) érvényesülne.

```
procedure olvas(var x, y: integer); forward;  // előre hivatkozás

procedure szamol1(x: integer; y: integer; var eredmeny:real);
var
   a, b: integer;
begin
   olvas(a,b);  // előbb használjuk, mint ahogy deklaráltuk
   //. . .
end;

procedure olvas(var x, y: integer);  // a teljes deklaráció
begin
  write('1. adat: '); readln(x);
  write('2. adat: '); readln(y);
end;
```

## 8.6 Rekurzív alprogramok

A matematikai algoritmusok egy része rekurzív módon, önmagát hívva működik. A rekurzív algoritmusok látványosak, és általában jól programozhatók. A Pascal nyelv is támogatja a rekurziót, hiszen lehetővé teszi, hogy valamely alprogram önmagát aktivizálja. Ne feledkezzünk meg arról, hogy minden rekurzív problémának létezik iteratív (ciklusos) megoldása, ami általában nehezebben programozható, azonban hatékonyabb és kevesebb memóriát igényel!

Példaként tekintsük a faktoriális iteratív és rekurzív definícióját (0! értéke 1)!

*iteratív megadás*

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

*rekurzív megadás*

$$n! = \begin{cases} 1, & ha\ n = 0 \\ n \cdot (n-1)!, & ha\ n > 0 \end{cases}$$

A rekurzív definíció felhasználásával 4! kiszámításának lépései:

```
4! = 4*3!
       3*2!
         2*1!
           1*0!
             1 = 4*3*2*1*1 = 24
```

Készítsünk rekurzív és nem rekurzív függvényt n! kiszámítására! (*faktorialis*)

```
program faktorialis;
var
  szam : integer;

function faktor_rekurziv(n: integer):longint;
  var
    fakt: longint;
  begin
      if n = 1 then fakt := 1
      else fakt := n*faktor_rekurziv(n-1);
    faktor_rekurziv:= fakt;
  end;

function faktor_nemrekurziv(n: integer):longint;
  var
    fakt: longint;
    i: integer;
  begin
    fakt:= 1;
    for i:=1 to n do
       fakt := fakt*i;
    faktor_nemrekurziv:= fakt;
end;

begin
  write('Kérek egy pozitív egész számot : ');
  readln(szam);
  writeln('A ',szam,' faktoriális rekurzívan    : ',
          faktor_rekurziv(szam));
  writeln('A ',szam,' faktoriális nem rekurzívan: ',
          faktor_nemrekurziv(szam));
  writeln;
  readln;
end.
```

*A program futásainak eredménye:*

```
Kérek egy pozitív egész számot: 6
A 6 faktoriális rekurzívan    : 720
A 6 faktoriális nem rekurzívan: 720

Kérek egy pozitív egész számot : 10
A 10 faktoriális rekurzívan    : 3628800
A 10 faktoriális nem rekurzívan: 3628800

Kérek egy pozitív egész számot : 12
A 12 faktoriális rekurzívan    : 479001600
A 12 faktoriális nem rekurzívan: 479001600
```

## 8.7 Paraméterek alapértéke

Az ***ObjFpc*** (és ***Delphi***) módokban alapértéket adhatunk a típusos konstans- és érték-paramétereknek az = jelet követő – a paraméter típusával kompatibilis – konstans értékkel. Az alapértékekkel ellátott paraméterek a formális paraméterlista végére kell, hogy kerüljenek. Ha egy formális paraméterlistán használtuk az alapértéket, akkor mindegyik azt követő paraméter esetén is ezt kell tennünk. Az alapérték megadását nem alkalmazhatjuk a többszörös típusmegadás ($x$, $y$ : real) során.

Az alprogram híváskor az alapértékkel bíró aktuális paraméterek balról jobbra haladva elhagyhatók.

Készítsünk háromszög területét számító függvényt, amely másképp számol derékszögű és általános háromszög esetén! Amennyiben a harmadik oldal értéke 0, akkor derékszögű háromszögről van szó. Legyen a 0 a harmadik oldal alapértéke! (*haromszog*)

```
program haromszog;
{$Mode ObjFpc}

function hterulet(a,b : real; c:real = 0) : real;
var s:real;
begin
  if c=0 then // derékszögű a háromszög
     result:=a*b/2
  else       // általános a háromszög
     begin
       s:=(a+b+c)/2; // számolás a Héron-képlettel
       result:=sqrt(s*(s-a)*(s-b)*(s-c));
     end;
end;

begin
  write('A derékszögű háromszög oldalai: 3, 4, 5 - területe: ');
  writeln(hterulet(3,4):8:3);   // hívás két argumentummal
  write('Az általános háromszög oldalai: 7, 7, 7 - területe: ');
  writeln(hterulet(7,7,7):8:3); // hívás három argumentummal
  readln;
end.
```

*A program futásának eredménye:*

```
A derékszögű háromszög oldalai: 3, 4, 5 - területe:    6.000
Az általános háromszög oldalai: 7, 7, 7 - területe:   21.218
```

## 8.8 Alprogramok több változatban (overloading)

A Free Pascalban azonos névvel (azonos céllal), de különböző formális paraméterlistával több eljárást és függvényt is készíthetünk. Mindegyik alprogramváltozat része lesz a programnak, és a fordító dönti el az argumentumlista alapján, hogy melyik hí-

vódik meg. Amennyiben nem találja meg a megfelelő változatot, fordítási hiba keletkezik.

Készítsünk valós alapot egész, illetve valós kitevőre emelő függvényeket! Használjuk az overloading mechanizmusát! (*hatvany2*)

```
program hatvany2;

function hatvany(a: real; k: integer): real;
var
   i: integer;
begin
   hatvany:= 1;
   for i:=1 to abs(k) do
     hatvany:= hatvany*a;
   if k < 0 then
     hatvany:= 1/hatvany;
end;

function hatvany(a: real; k: real): real;
begin
   hatvany:= exp(k*ln(a));
end;

begin
  write('12.23^12   = ');
  writeln(hatvany(12.23, 12):18:3);

  write('12.23^12.0 = ');
  writeln(hatvany(12.23, 12.0):18:3);
  readln;
end.
```

*A program futásának eredménye:*

```
12.23^12   = 11197407524272.840
12.23^12.0 = 11197407524272.850
```

Megjegyezzük, hogy ***Delphi*** módban az átdefiniált alprogramváltozatok fejlécét az **overload** foglalt szóval kell zárnunk.

## 8.9 Operátorok túlterhelése (overloading)

A Free Pascal lehetővé teszi, hogy bizonyos műveleti jelekkel kijelölt műveleteket elvégezzünk a saját magunk által definiált típusú adatokon. Ehhez a műveleteket függvényként kell értelmezni, melynek paraméterei az operandusok, értéke pedig a művelet elvégzésének eredménye:

```
operator műveleti_jel(paraméterlista) eredmény: típus;
 // lokális deklarációs rész
 // label, const, type, var, procedure, function deklarációk
begin
    // a műveleti függvény törzse - végrehajtható utasítások
    eredmény := kifejezés;
end;
```

Megjegyezzük, hogy ***ObjFpc*** (és ***Delphi***) módokban az ***eredmény*** azonosító helyett a **result** azonosítót is használjuk.

Az átdefiniálható Pascal műveleti jeleket három csoportba sorolhatjuk.

| | |
|---|---|
| Az értékadó operátor: | := |
| Aritmetikai műveletek: | összeadás (+), kivonás (-), szorzás (*), osztás(/) és a hatványozás (**) (a ** operátort a Pascal nem ismeri), |
| Összehasonlító műveletek: | =, <, <=, >, >= (a <> nem szerepel a listában). |

Értékadó művelet esetén az eredmény típusának meg kell egyeznie az értékadás bal oldalán álló változó típusával, míg az operátorfüggvény egyetlen paramétere megkapja a jobb oldalon álló kifejezés értékét. A Free Pascal jelen változatában az értékadás két oldalának eltérő típusúnak kell lennie.

Az aritmetikai és a logikai műveletek két operandussal rendelkeznek, melyek közül a bal oldalit az operátorfüggvény első paramétere, míg a jobb oldalit a második paramétere képviseli. A kifejezés értéke pedig az eredmény típusával egyezik meg. Összehasonlító műveletek esetén az eredmény csak ***boolean*** típusú lehet

A ** operátor túlterhelésével valósítsuk meg a valós alapot egész, illetve valós kitevőre emelő műveleteket! (*hatvany3*)

```
program hatvany3;

operator **(a: real; k: integer) hatvany: real;
var
   i: integer;
begin
   hatvany:= 1;
   for i:=1 to abs(k) do
     hatvany:= hatvany*a;
   if k < 0 then
     hatvany:= 1/hatvany;
end;

operator **(a: real; k: real) hatvany: real;
begin
   hatvany:= exp(k*ln(a));
end;
```

```
begin
  write('12.23^12    = ');
  writeln(12.23**12:18:3);

  write('12.23^12.0 = ');
  writeln(12.23**12.0:18:3);
  readln;
end.
```

*A program futásának eredménye:*

```
12.23^12    = 11197407524272.840
12.23^12.0 = 11197407524272.850
```

Három térbeli koordinátát tároló rekordhoz rendeljünk műveleteket mindhárom operátorcsoportból! (*terbelivektor*)

```
program TerbeliVektor;
uses sysutils;

// térbeli vektor
type
  vektor3D = record
    x, y, z : integer;
  end;

// azonos koordináták
operator := (a : integer) v : vektor3D;
begin
  v.x := a;
  v.y := a;
  v.z := a;
end;

// vektor szorzása számmal
operator * (r : integer; v1 : vektor3D) v2 : vektor3D;
begin
  v2.x := v1.x * r;
  v2.y := v1.y * r;
  v2.z := v1.z * r;
end;

// két vektor skaláris szorzata
operator * (v1 : vektor3D; v2 : vektor3D) s : integer;
begin
  s:= v1.x*v2.x + v1.y*v2.y + v1.z * v2.z;
end;

// két vektor vektoriális  szorzata
operator ** (v1 : vektor3D; v2 : vektor3D) v3 : vektor3D;
begin
  v3.x:= v1.y*v2.z-v2.y*v1.z;
  v3.y:= -(v1.x*v2.z-v2.x*v1.z);
  v3.z:= v1.x*v2.y-v2.x*v1.y;
end;
```

```
// két vektor azonosságának vizsgálata
operator = (v1 : vektor3D; v2 : vektor3D) s : boolean;
begin
  s:= (v1.x=v2.x) and (v1.y=v2.y) and (v1.z=v2.z);;
end;

// vektor átalakítása szöveggé a kiíráshoz
function svektor(const v:vektor3D) : string;
begin
  svektor:='('+inttostr(v.x)+','+inttostr(v.y)+
           ','+inttostr(v.z)+')';
end;

var
  a, b : vektor3D;
  c : vektor3D = (x:7;y:12;z:23);
begin
  a := 1;
  writeln('a = ',svektor(a));
  b := c;
  writeln('c = ',svektor(b));
  b := 2*c;
  writeln('2*c = ',svektor(b));
  writeln('a*c = ',a*c);
  b := a**c;
  writeln('a**c = ',svektor(b));
  if a=b then
    writeln('a azonos b-vel')
  else
    writeln('a nem azonos b-vel');
  readln;
end.
```

*A program futásának eredménye:*

```
a = (1,1,1)
c = (7,12,23)
2*c = (14,24,46)
a*c = 42
a**c = (11,-16,5)
a nem azonos b-vel
```

## 8.10 Alprogramok készítése különböző típusú paraméterekkel

Az alábbi részben példák segítségével áttekintjük az alprogramok paraméterezésével, illetve a függvények értékével kapcsolatos ismereteket. Az egyes alfejezetekben – a teljesség igénye nélkül – megnézzük az

- egyszerű típusok használatát – a **string** típusokat is ide sorolva,
- a tömb típusú paraméterek alkalmazását, külön kitérve az ún. nyitott tömbparaméterekre,
- a rekord típusú paraméterekkel történő programozást,
- valamint speciális paraméterek és alprogramok felhasználását.

### 8.10.1 Egyszerű típusú paraméterek és függvényértékek

Egyszerű típusok esetén az értékparaméterek is hatékonyan használhatók, mivel a típushoz kis memóriaterületek tartoznak. (Kivételt talán csak a rövid sztringek képeznek, melyek hossza legfeljebb 255 karakter.)

Készítsünk függvényt, amely igaz értéket ad vissza, ha a paraméterben kapott egész érték páros, és hamisat, ha páratlan! (*fvparitas*)

A *paros()* függvény fejléce:

```
function paros(x : integer):boolean;
```

*A feladat megoldása:*

```
program fvparitas;
var
  d: integer;

function paros(x : integer):boolean;
begin
   paros := not odd(x);
end;

begin
  write('Egész szám: '); readln(d);
  if paros(d) then
    writeln(d,' páros szám')
  else
    writeln(d,' páratlan');
  readln;
end.
```

*A program futásainak eredményei:*

```
Egész szám: 13
13 páratlan

Egész szám: 12
12 páros szám
```

Írjunk programot, amely beolvassa egy kör sugarát, és kiszámítja a kör területét és kerületét! (*korszamitas*)

A kör sugarának beolvasása eljárással:

```
procedure olvas(var r: real);
```

A kör területének és kerületének számítása függvényekkel:

```
function kor_ter(r:real): real;
function kor_ker(r:real): real;
```

*A feladat megoldása:*

```
program korszamitas;

procedure olvas(var r: real);
begin
  write('A kör sugara: '); readln(r);
end;

function kor_ter(r:real): real;
begin
  kor_ter:= sqr(r)*pi;
end;

function kor_ker(r:real): real;
begin
  kor_ker:=2*r*pi;
end;

var
  sugar, ter, ker : real;
begin
  olvas(sugar);
  ter:= kor_ter(sugar);
  ker:= kor_ker(sugar);
  writeln('A kör területe: ', ter:6:2);
  writeln('A kör kerülete: ', ker:6:2);
  readln;
end.
```

*A program futásának eredménye:*

```
A kör sugara: 1
A kör területe:   3.14
A kör kerülete:   6.28
```

Készítsünk programot, amely egy karaktersorozatról megállapítja, hogy egész számot tárol-e! A tesztelő főprogramban egyaránt használjunk hosszú és rövid sztringeket! (*egeszstr*)

A megoldásban a **string** típust használtuk függvényértékként és bemenő (érték, **const**) paraméterként. Ilyen esetekben a különböző típusú karaktersorozatok átalakítását a fordító minden további nélkül elvégzi. (Ez azonban nem működik változóparaméterekkel.)

```
program egeszstr;
// A szám beolvasása karakteres formában
function beolvas() : string;
var
  s : string;
begin
  write('Kérem az egész számot: '); readln(s);
  beolvas := s;
end;
```

```
// A karaktersorozatban tárolt egész szám ellenőrzése
function szam_e(const s: string) : boolean;
var
  i, p : integer;
begin
  szam_e:=true;
  // a bevezető szóközök átlépése
  p:=0;
  repeat
    inc(p);
  until s[p]<>#32;
  // a szám előtt állhat +,- előjel, átlépjük
  if s[p] in ['+','-'] then inc(p);
  // csak decimális jegyek következhetnek
  for i:=p to length(s) do
    if not(s[i] in ['0'..'9']) then
      begin
        szam_e:=false;
        break;
      end;
end;

var
  s1 : ansistring;
  s2 : shortstring;
  s3 : string[12];
begin
  s1 := beolvas();
  s2 := beolvas();
  s3 := beolvas();
  if szam_e(s1) then
     writeln(s1, ' egész szám')
  else
     writeln(s1, ' nem egész szám');
  if szam_e(s2) then
     writeln(s2, ' egész szám')
  else
     writeln(s2, ' nem egész szám');
  if szam_e(s3) then
     writeln(s3, ' egész szám')
  else
     writeln(s3, ' nem egész szám');
  readln;
end.
```

*A program futásának eredménye:*

```
Kérem az egész számot: -123
Kérem az egész számot:    2312
Kérem az egész számot: +79.1234
-123 egész szám
   2312 egész szám
+79.1234 nem egész szám
```

Írjunk programot, amely a beolvasott szöveget titkosítja, méghozzá úgy, hogy minden betűt a rákövetkezővel helyettesíti! Készítsük el a dekódoló alprogramot is! A tesztelő főprogramban egyaránt használjunk hosszú és rövid sztringeket! (*titkosstr*)

Mivel a változóparamétereknél szigorú típusellenőrzést hajt végre a fordító, csak a túlterhelés (*overloading*) alkalmazásával jutunk olyan megoldáshoz, amely tetszőleges típusú karaktersorozattal hívható.

```
program titkosstr;

// A szöveg beolvasását végző túlterhelt eljárások
procedure beolvas(var s:ansistring);
begin
  write('Kérek egy szöveget: '); readln(s);
end;

procedure beolvas(var s:shortstring);
begin
  write('Kérek egy szöveget: '); readln(s);
end;

// Titkosítás:  A->B, B->C, ... Y->Z, Z->A
procedure titkosit(var s:ansistring);
var
  i : integer;
begin
  for i:=1 to length(s) do
    if s[i] in ['A'..'Z', 'a'..'z'] then
       if upcase(s[i])='Z' then
         dec(s[i],25)
       else
         inc(s[i]);
end;

procedure titkosit(var s:shortstring);
var
  i : integer;
begin
  for i:=1 to length(s) do
    if s[i] in ['A'..'Z', 'a'..'z'] then
       if upcase(s[i])='Z' then
         dec(s[i],25)
       else
         inc(s[i]);
end;

// Dekódolás:  A->Z, B->A, ... Y->X, Z->Y
procedure dekodol(var s:ansistring);
var
  i : integer;
begin
  for i:=1 to length(s) do
```

```
    if s[i] in ['A'..'Z', 'a'..'z'] then
       if upcase(s[i])='A' then
         inc(s[i],25)
       else
         dec(s[i]);
end;

procedure dekodol(var s:shortstring);
var
  i : integer;
begin
  for i:=1 to length(s) do
    if s[i] in ['A'..'Z', 'a'..'z'] then
       if upcase(s[i])='A' then
         inc(s[i],25)
       else
         dec(s[i]);
end;

var
  s1:ansistring;      s2:shortstring;      s3:string[35];
begin
  beolvas(s1);        beolvas(s2);         beolvas(s3);
  titkosit(s1);       titkosit(s2);        titkosit(s3);
  writeln(s1);        writeln(s2);         writeln(s3);
  dekodol(s1);        dekodol(s2);         dekodol(s3);
  writeln(s1);        writeln(s2);         writeln(s3);
  readln;
end.
```

*A program futásának eredménye:*

```
Kérek egy szöveget: Csak azt kell tudnod, eltel a maban!
Kérek egy szöveget: Csak azt kell hinned, nem volt hiaba!
Kérek egy szöveget: Nezd az ebredot, visszater!
Dtbl bau lfmm uveope, fmufm b nbcbo!
Dtbl bau lfmm ijoofe, ofn wpmu ijbcb!
Ofae ba fcsfepu, wjttabufs!
Csak azt kell tudnod, eltel a maban!
Csak azt kell hinned, nem volt hiaba!
Nezd az ebredot, visszater!
```

### 8.10.2 Tömbtípusú paraméterek és függvényértékek

Először megnézzük a hagyományos megoldást, amikor típusdefinícióval biztosítjuk, hogy a formális és az aktuális paraméterek azonos típusúak legyenek. Ebben az esetben a tömbök aktuális elemszámát szintén át kell adnunk az alprogramoknak. Ezt követően pedig a kevésbé típusfüggő, nyitott tömbparaméterekkel foglalkozunk. Felhívjuk a figyelmet arra, hogy tömbök esetén az érték-, a konstans és a változóparaméter egyaránt alkalmazható, azonban értékparaméter esetén nagymennyiségű adat másolódhat. Megjegyezzük, hogy Free Pascalban a függvényérték is lehet tömbtípusú.

Írjunk programot, amely adott darabszámú, egész típusú adatottal tölt fel egy tömböt! Ezt követően pedig kiszámítja az adatok átlagát, és megszámlálja a pozitív, a negatív és a zérusértékű elemeket! (*tombatlag*)

Az alábbi típusdefiníciót és deklarációkat alkalmazzuk:

```
type tomb = array[1..20] of integer;
var
  y : tomb;
  y_db, ypoz, yneg, yzero: integer;
  yatlag: real;
```

Az adatokat eljárással olvassuk be:

```
procedure TOlvas(var x : tomb; var n: integer);
```

Az adatok átlagát függvénnyel határozzuk meg:

```
function TAtlag(const x: tomb; n: integer):real;
```

A pozitív, negatív és zérus adatok számlálását eljárással oldjuk meg:

```
procedure TKeres(const x: tomb; n: integer;
                 var poz, neg, zero: integer);
```

*A feladat megoldása:*

```
program tombatlag;
type
    tomb = array[1..20] of integer;

procedure TOlvas(var x : tomb; var n: integer);
var
    i: integer;
begin
    repeat
      write('Adatok száma: '); readln(n);
    until n in [1..20];
    for i:=1 to n do
    begin
      write(i:2,'. adat: '); readln(x[i]);
    end;
end;

function TAtlag(const x: tomb; n: integer):real;
var
    i: integer;
    atlag: real;
begin
    atlag:= 0;
    for i:=1 to n do
      atlag:= atlag+x[i];
    TAtlag:= atlag/n;
end;
```

```
procedure TKeres(const x: tomb; n: integer;
                 var poz, neg, zero: integer);
var
    i: integer;
begin
    poz:= 0; neg:= 0; zero:= 0;
    for i:=1 to n do
    begin
      if x[i] > 0 then inc(poz);
      if x[i] < 0 then inc(neg);
      if x[i] = 0 then inc(zero);
    end;
end;

var
    y : tomb;
    y_db, ypoz, yneg, yzero: integer;
    yatlag: real;
begin
    TOlvas(y,y_db);
    yatlag:= TAtlag(y, y_db);
    TKeres(y, y_db, ypoz, yneg, yzero);
    writeln('Az adatok átlaga: ',yatlag:6:2);
    writeln('A pozitív adatok száma: ',ypoz);
    writeln('A negatív adatok száma: ',yneg);
    writeln('A zérus adatok száma  : ',yneg);
    readln;
end.
```

*A program futásának eredménye:*

```
Adatok száma: 5
 1. adat: 2
 2. adat: -4
 3. adat: 6
 4. adat: 0
 5. adat: 7
Az adatok átlaga:   2.20
A pozitív adatok száma: 3
A negatív adatok száma: 1
A zérus adatok száma  : 1
```

Pascal program segítségével rendezzünk névsorba megadott számú nevet! A rendezést nyitott tömb paraméterrel oldjuk meg! (*nevsor*)

A nyitott tömbparaméter lehetőséget biztosít arra, hogy olyan alprogramokat készítsünk, melyek hívása nem függ az egydimenziós paramétertömb méretétől. Ehhez a formális paramétertömböt az

**array of** *elemtípus*

típussal kell deklarálnunk. Az eljáráson és a függvényen belül úgy használhatjuk a nyitott tömbparamétert, mintha az

```
array [0..N-1] of elemtípus
```

típusú lenne, ahol *N* az aktuális paraméterként átadott tömb elemeinek száma. Az alprogramon belül a ***high()*** függvénnyel kérdezhetjük le a nyitott tömbparaméter utolsó elemének indexét. (A ***low()*** függvény *0* értéket ad, és a ***sizeof()*** az aktuális paramétertömb bájtokban kifejezett méretét szolgáltatja.)

```
program nevsor;

const
  nszam = 6;    // a nevek száma
type
  TNevek = array[1..nszam] of string;

// A beolvasó eljárás hagyományos paraméterezéssel
procedure beolvas(var nevek : TNevek);
var
  i : integer;
begin
  for i:=1 to nszam do
    begin
      write(i:3,'. név : ');
      readln(nevek[i]);
    end;
end;

// A rendező eljárás nyitott tömbparaméterrel
procedure rendez(var nevek : array of string);
var
  i, j : integer;
  nev  : string;
begin
  for i:=0 to high(nevek)-1 do
    for j:=i+1 to high(nevek) do
       if nevek[i]>nevek[j] then // csere
          begin
            nev       := nevek[i];
            nevek[i] := nevek[j];
            nevek[j] := nev;
          end;
end;

// A megjelenítés szintén nyitott tömbparamétereket használ
procedure kiir(const nevek : array of string);
var
  i : integer;
begin
  for i:=0 to high(nevek) do
     writeln(nevek[i]);
end;

var
  nevlista : TNevek;
```

```
begin
  beolvas(nevlista);
  rendez(nevlista);
  kiir(nevlista);
  readln;
end.
```

*A program futásának eredménye:*

```
  1. név : Natália
  2. név : Anna
  3. név : Iván
  4. név : Adrienn
  5. név : Júlia
  6. név : Léna
Adrienn
Anna
Iván
Júlia
Léna
Natália
```

Készítsünk programot, amely véletlen, egész értékekkel tölt fel egy adott típusú kétdimenziós (négyzetes) tömböt! A feladatot egyaránt oldjuk meg hagyományos módon eljárással, illetve a tömböt értékként visszaadó függvénnyel! (*rndmatrix*)

```
program rndmatrix;

const
  meret = 5;    // a mátrix sorainak és oszlopainak száma
type
  TMatrix = array[1..meret, 1..meret] of integer;

// A feltöltő eljárás hagyományos paraméterezéssel
procedure pfeltolt(var mat : TMatrix);
var
  i, j : integer;
begin
  for i:=1 to meret do
    for j:=1 to meret do
      mat[i,j] := random(2006);
end;

// A feltöltő függvény tömbtípusú függvényértékkel
function ffeltolt(): TMatrix;
var
  i, j : integer;
begin
  for i:=1 to meret do
    for j:=1 to meret do
      ffeltolt[i,j] := random(2006);
end;
```

```
// A megjelenítés hagyományos tömbparamétert használ
procedure kiir(const matrix : TMatrix);
var
  i, j : integer;
begin
  for i:=1 to meret do
    begin
      for j:=1 to meret do
        write(matrix[i,j]:6);
      writeln;
    end;
end;

var
  matrix1, matrix2 : TMatrix;
begin
  randomize;
  pfeltolt(matrix1);
  matrix2 := ffeltolt();
  kiir(matrix1);
  writeln;
  kiir(matrix2);
  readln;
end.
```

*A program futásának eredménye:*

```
  1391   554   675   419   722
   103  1488  1555   320   570
  1761  1307    27   280  1647
   775  1404   226  1611  1829
   903   173   706  1295  1844

  1132  1186   797   681   807
  1446  1280  1858  1275  1377
  1351  1941  1123  1264   550
   804  1172   781  1915  1737
  1978  1204  1527   841   813
```

Írjunk programot, legfeljebb 5x5-ös mátrix determinánsának rekurzív meghatározására! A teszteléshez használjunk konstans tömböket! (*rekdet*)

A megoldás lényege, hogy az *N*-ed rendű determináns meghatározását *N* darab (*N-1*)-ed rendű determináns számítására vezetjük vissza. Tesszük ezt mindaddig, míg el nem jutunk a másodrendű determinánsokig. A számítás során a hívások fastruktúrába láncolódnak, ami sok futásidőt és memóriát igényel.

```
program rekdet;

type
   TMatrix = array[1..5, 1..5] of real;
```

```
procedure MatrixKiir(const m : TMatrix; n:integer);
var
  i, j : integer;
begin
  for i:=1 to n do
    begin
      for j:=1 to n do
        write(m[i,j]:6:2);
      writeln;
    end;
end;

function Det(const m:TMatrix; n : integer) : real;
var
  i, j, k, q : integer;
  s : real = 0;
  x : TMatrix;
begin
  if n=2 then
     exit(m[1][1]*m[2][2]-m[2][1]*m[1][2]);
  for k:=1 to n do  // n darab aldeterminánsa
    begin
        // az almátrixok létrehozása
        for i:=2 to n do          // sorok
         begin
           q := 1;
           for j:=1 to n do       // oszlopok
            if j<>k then
             begin
               x[i-1][q] := m[i][j];
               inc(q);
             end;
         end;
        s:= s + (2*(k mod 2) - 1) * m[1][k] * Det(x,n-1); // rekurzió
    end;
  Det := s;
end;

const
  m2 : TMatrix = (( 1, 2, 0, 0, 0),
                  ( 2, 3, 0, 0, 0),
                  ( 0, 0, 0, 0, 0),
                  ( 0, 0, 0, 0, 0),
                  ( 0, 0, 0, 0, 0));

  m3 : TMatrix = (( 1, 2, 3, 0, 0),
                  ( 2, 1, 1, 0, 0),
                  ( 1, 0, 1, 0, 0),
                  ( 0, 0, 0, 0, 0),
                  ( 0, 0, 0, 0, 0));

  m4 : TMatrix = (( 2, 0, 4, 3, 0),
                  (-1, 2, 6, 1, 0),
                  (10, 3, 4,-2, 0),
                  ( 2, 1, 4, 0, 0),
                  ( 0, 0, 0, 0, 0));
```

```
begin
  MatrixKiir(m2,2);
  writeln('Det(m2) = ',Det(m2,2):8:3);
  writeln;
  MatrixKiir(m3,3);
  writeln('Det(m3) = ',Det(m3,3):8:3);
  writeln;
  MatrixKiir(m4,4);
  writeln('Det(m4) = ',Det(m4,4):8:3);
  readln;
end.
```

*A program futásának eredménye:*

```
  1.00  2.00
  2.00  3.00
Det(m2) =   -1.000

  1.00  2.00  3.00
  2.00  1.00  1.00
  1.00  0.00  1.00
Det(m3) =   -4.000

  2.00  0.00  4.00  3.00
 -1.00  2.00  6.00  1.00
 10.00  3.00  4.00 -2.00
  2.00  1.00  4.00  0.00
Det(m4) =  144.000
```

### 8.10.3 Rekordtípusú paraméterek és függvényértékek

A rekordtípust (érték-, konstans és változó-) paraméterek valamint függvényértékek deklarálására egyaránt használhatjuk. (Megjegyezzük, hogy rekordok kezeléséhez hasonlóan készíthetünk **object** és **class** típusú objektumokat feldolgozó eljárásokat és függvényeket.)

Készítsünk Pascal nyelvű programot, amely beolvassa egy téglatest három oldalának hosszát, és kiszámítja a téglatest felszínét és térfogatát! A téglatest adatait tároljuk rekordban! (*teglarekord*)

Az alábbi típusdefiníciót és deklarációt használjuk:

```
type teglatest = record
                   a, b, c: real;
                   terf, felsz: real;
                 end;
var
  tg : teglatest;
```

A *teglatest* típusú rekord *a,b,c* adatmezőit eljárással olvassuk be:

```
    procedure Olvas(var t : teglatest);
```

A téglatest felszínét és térfogatát a *Szamol()* eljárásban határozzuk meg:

```
procedure Szamol(var t: teglatest);
```

A téglatest felszínét és térfogatát szintén eljárással jelenítjük meg:

```
procedure Kiir(t: teglatest);
```

*A feladat megoldása:*

```
program teglarekord;

type teglatest = record
                   a, b, c: real;
                   terf, felsz: real;
                 end;

procedure Olvas(var t : teglatest);
begin
    writeln('Téglatest adatai ');
    write('A oldal: '); readln(t.a);
    write('B oldal: '); readln(t.b);
    write('C oldal: '); readln(t.c);
end;

procedure Szamol(var t: teglatest);
begin
    with t do
      begin
        terf:= a * b * c;
        felsz:= 2*(a * b + b * c + a * c)
      end;
end;

procedure Kiir(t: teglatest);
begin
     writeln('Téglatest felszíne : ',t.felsz:6:2);
     writeln('Téglatest térfogata: ',t.terf:6:2);
end;

var
  tg : teglatest;
begin
    Olvas(tg);
    Szamol(tg);
    Kiir(tg);
    readln;
end.
```

*A program futásának eredménye:*

```
Téglatest adatai
A oldal: 1
B oldal: 2
C oldal: 3
Téglatest felszíne :  22.00
Téglatest térfogata:   6.00
```

Írjunk programot, amely adott elemszámú rekordtömböt feltölt adatokkal! Számítsuk ki a *doboz* típusú elemek össztérfogatát, keressük ki a legnagyobb súlyú és a legkisebb fajsúlyú dobozt, valamint írassuk ki a dobozok adatait! (*dobozrek*)

A megoldásban az alábbi deklarációkat használjuk:

```
type  doboz = record
                tartalom: string;
                a, b, c: real;  // oldalhosszak
                terfogat, fajsuly, suly : real;
              end;
      dobozok = array[1..10] of doboz;
var
  d : dobozok;
  db: integer;
  max_suly, ossz_terfogat, min_fajsuly: real;
  tartalom: string;
```

A *dobozok* típusú rekord *a,b,c* adatmezőit eljárással olvassuk be:

```
procedure DobozokOlvas(var dk: dobozok; var n: integer);
```

A *MaxSulyKeres()* eljárással keressük ki a legnagyobb súlyú és a legkisebb fajsúlyú dobozt:

```
procedure MaxSulyKeres(const dk: dobozok; n:integer;
                       var MaxSuly, MinFajsuly:real);
```

A dobozok össztérfogatát meghatározó függvény:

```
function OsszTerfogat(const dk: dobozok; n: integer): real;
```

A dobozok adatait megjelenítő eljárás:

```
procedure Kiir(const dk: dobozok; n: integer);
```

*A feladat megoldása:*

```
program dobozrek;

const maxdoboz = 10;

type  doboz = record
                tartalom: string[12];
                a, b, c: real;
                terfogat, fajsuly, suly : real;
              end;
      dobozok = array[1.. maxdoboz] of doboz;

var
  d : dobozok;
  db: integer;
  max_suly, ossz_terfogat, min_fajsuly: real;
  tartalom: string;
```

```
procedure DobozokOlvas(var dk: dobozok; var n: integer);
var
     i: integer;
begin
    repeat
       write('Dobozok darabszáma: '); readln(n);
    until n in [1..maxdoboz];
    writeln('A dobozok adatai');
    for i:=1 to n do
      with dk[i] do
        begin
          write('A oldala : '); readln(a);
          write('B oldala : '); readln(b);
          write('C oldala : '); readln(c);
          write('A fajsúly: '); readln(fajsuly);
          write('Tartalom : '); readln(tartalom);
          writeln;
        end;
end;

procedure Szamol(var dk: dobozok; n: integer);
var
     i: integer;
begin
    for i:=1 to n do
    begin
      dk[i].terfogat := dk[i].a * dk[i].b * dk[i].c;
      dk[i].suly := dk[i].terfogat*dk[i].fajsuly;
    end;
end;

procedure MaxSulyKeres(const dk: dobozok; n:integer;
                       var MaxSuly, MinFajsuly:real);
var
    i: integer;
begin
    MaxSuly:= dk[1].suly;
    MinFajsuly := d[1].fajsuly;
    for i:=2 to n do
    begin
      if dk[i].suly > MaxSuly then MaxSuly:= dk[i].suly;
      if dk[i].fajsuly < MinFajsuly then MinFajsuly:= dk[i].fajsuly;
    end;
end;

function OsszTerfogat(const dk: dobozok; n: integer): real;
var
     i: integer;
     osszt: real;
begin
     osszt:= 0;
     for i:=1 to n do
       osszt:= osszt+dk[i].terfogat;
     OsszTerfogat := osszt;
end;
```

```
procedure TartalomKeres(const dk: dobozok; n: integer;
                        Doboztartalom: string);
var
    i: integer;
    talalt: boolean;
begin
    talalt := false;
    for i:=1 to n do
    begin
      if dk[i].tartalom = Doboztartalom then
       begin
         talalt := true;
         writeln;
         writeln('A doboz adatai');
         writeln('Tartalma : ',dk[i].tartalom);
         writeln('Oldalai  : ', dk[i].a:6:2, ' ',dk[i].b:6:2,
                 ' ',dk[i].c:6:2);
         writeln('Térfogata: ',dk[i].terfogat:6:2);
         writeln('Súlya    : ',dk[i].suly:6:2);
         writeln('Fajsúlya : ',dk[i].fajsuly:6:2);
         writeln;
         break;
       end;
    end;
    if not talalt then
     begin
       writeln(Doboztartalom,' nem található!');
       writeln;
     end;
end;

procedure Kiir(const dk: dobozok; n: integer);
var
     i: integer;
begin
     writeln;
     writeln('A dobozadatok megjelenítése:');
     writeln;
     writeln('   a      b      c     térfogat          súly' +
             '  fajsúly     tartalom');
     for i:=1 to n do
      with dk[i] do
         begin
            write(a:6:1);
            write(b:6:1);
            write(c:6:1);
            write(terfogat:10:2);
            write(suly:13:2);
            write(fajsuly:9:1);
            writeln(tartalom:13);
         end;
end;
```

```
begin
    DobozokOlvas(d,db);
    Szamol(d,db);
    MaxSulyKeres(d,db,max_suly, min_fajsuly);
    ossz_terfogat:= OsszTerfogat(d,db);
    write('Tartalom keresése: '); readln(tartalom);
    TartalomKeres(d,db,tartalom);
    writeln('Legnagyobb súlyú doboz  : ',max_suly:6:2);
    writeln('Legkisebb fajsúlyú doboz: ',min_fajsuly:6:2);
    Kiir(d,db);
    readln;
end.
```

*A program futásának eredménye:*

```
Dobozok darabszáma: 3
A dobozok adatai
A oldala : 30
B oldala : 40
C oldala : 25
A fajsúly: 10
Tartalom : anyag1

A oldala : 35
B oldala : 40
C oldala : 30
A fajsúly: 15
Tartalom : anyag2

A oldala : 30
B oldala : 40
C oldala : 25
A fajsúly: 12
Tartalom : anyag3

Tartalom keresése: anyag1

A doboz adatai
Tartalma : anyag1
Oldalai  :  30.00  40.00  25.00
Térfogata: 30000.00
Súlya    : 300000.00
Fajsúlya :  10.00

Legnagyobb súlyú doboz  : 630000.00
Legkisebb fajsúlyú doboz:  10.00

A dobozadatok megjelenítése:
   a     b     c    térfogat         súly  fajsúly     tartalom
 30.0  40.0  25.0  30000.00    300000.00     10.0       anyag1
 35.0  40.0  30.0  42000.00    630000.00     15.0       anyag2
 30.0  40.0  25.0  30000.00    360000.00     12.0       anyag3
```

### 8.10.4 Speciális paraméterek és alprogramok

Ebben a részben bemutatunk néhány, ritkábban használt paramétertípust, valamint további példákat hozunk a rekurzió alkalmazására.

### *8.10.4.1 Típus nélküli paraméterek*

A Free Pascal lehetővé teszi, hogy a formális paraméterlistán elhagyjuk a változó- vagy a konstans paraméter típusát. Az alprogram hívásakor a típus nélküli paraméternek megfelelő aktuális paraméter tetszőleges típusú változó lehet. A típus nélküli paramétereket akkor használunk, amikor az alprogram működése nem típusfüggő (a változók területének másolása, nullázása stb.).

Készítsünk programot, amely lehetővé teszi tetszőleges típusú változó területének lenullázását, valamint két tetszőleges változó közötti értékadást! (*nullazmasol*)

Mivel a változókat típus nélküli paraméterben fogadjuk, egy másik paraméterben a változók bájtokban kifejezett méretét is át kell adnunk. Az eljáráson belül a változók területét típuskonverzió segítségével érjük el.

```
program nullazmasol;

procedure nullaz(var v; meret:integer);
type
    bt = array[1..2*maxint] of byte;
var
    i : integer;
begin
   for i:=1 to meret do
     bt(v)[i]:=0;
end;

procedure masol(var vbal; const vjobb; meret:integer);
type
    bt = array[1..2*maxint] of byte;
var
    i : integer;
begin
   for i:=1 to meret do
     bt(vbal)[i]:=bt(vjobb)[i];
end;

var
  a : array[1..4, 1..2] of real;
  b : record
        s : string[12];
        x : real;
      end;
  c, d : real;

begin
  nullaz(a, sizeof(a));
  nullaz(b, sizeof(b));
  nullaz(c, sizeof(c));
  writeln('c=',c:8:2);
  d := 12.23;
```

```
  masol(c,d,sizeof(d));
  writeln('c=',c:8:2);
  readln;
end.
```

*A program futásának eredménye:*

```
c=     0.00
c=    12.23
```

### *8.10.4.2 Eljárás- és függvényparaméterek*

Valamely alprogram paraméterlistáján függvény és eljárás is szerepelhet. Az összetett típusú paraméterekhez hasonlóan, a függvény- és eljárás-paraméter deklarálásához, létre kell hozni a megfelelő függvény-, illetve eljárástípust:

```
type
   függvénytípus = function(paraméterek) : típus;
   eljárástípus = procedure(paraméterek);
```

Free Pascalban az aktuális paraméter helyén a függvény/eljárás címét kell átadnunk, melyet a @ operátorral nyerünk:

```
@függvénynév
@eljárásnév
```

Készítsünk programot, amelyben egy eljárás a paraméterként kapott függvény értékeiből, adott intervallumon, táblázatot készít! (*fvtabla*)

```
program fvtabla;
type
   TFuggveny = function(x:real): real;

function fv1(x:real): real;
begin
   fv1 := 2*cos(x)+sin(x/3);
end;

function fv2(x:real): real;
begin
   fv2 := 12*sqr(x/2)/ln(3*x);
end;

// Az értéktáblát megjelenítő eljárás
procedure tablazat(x1, x2, dx : real; f:TFuggveny;
                   szoveg: string);
var
   x : real;
begin
   writeln('   x         ',szoveg);
   x:=x1;
```

```
   while x<=x2 do
     begin
        writeln(x:6:2,f(x):12:4);
        x:=x+dx;
     end;
end;

var
   fejlec1 : string = '2*cos(x)+sin(x/3)';
   fejlec2 : string = '12*sqr(x/2)/ln(3*x)';
begin
   writeln('Táblázatkészítés függvényparaméterrel');
   writeln;
   writeln('   Első táblázat ');
   tablazat(0.2,1.2,0.1,@fv1,fejlec1); writeln;
   writeln('   Második táblázat ');
   tablazat(1.2,2.3,0.15,@fv2,fejlec2);
   readln;
end.
```

*A program futásának eredménye:*

```
Táblázatkészítés függvényparaméter

   Első táblázat
   x         2*cos(x)+sin(x/3)
  0.20       2.0268
  0.30       2.0105
  0.40       1.9751
  0.50       1.9211
  0.60       1.8493
  0.70       1.7609
  0.80       1.6569
  0.90       1.5387
  1.00       1.4078
  1.10       1.2657
  1.20       1.1141

   Második táblázat
   x         12*sqr(x/2)/ln(3*x)
  1.20       3.3725
  1.35       3.9089
  1.50       4.4878
  1.65       5.1066
  1.80       5.7638
  1.95       6.4579
  2.10       7.1881
  2.25       7.9535
```

### *8.10.4.3 Rekurzív alprogramok*

Mint említettük, a rekurzió igen érdekes feladat-megoldási lehetőség. Eltekintve a programozása során felmerülő problémáktól (memóriaigény, futási idő stb.), szép, jól strukturált programokat készíthetünk a felhasználásával.

Készítsünk függvényt, amely kiszámolja, a rekurzív módon megadott számsorozat n-edik elemét! (*reksorozat*)

A sorozat rekurzív definíciója:

$a_1 = 3$

$a_n = 2*a_{n-1} + 5$, ha $n>1$

```
program reksorozat;
function an(n: integer): longint;
var
  x: longint;
begin
  if n = 1 then
       x := 3
  else
       x := 2 * an(n-1) + 5;
  an:= x;
end;

var
  elem: longint;
begin
  write('Hányadik elem: ');
  readln(elem);
  writeln('a(z) ',elem,'. elem: ',an(elem));
  readln;
end.
```

*A program futásainak eredménye:*

```
Hányadik elem: 1
a(z) 1. elem: 3
Hányadik elem: 12
a(z) 12. elem: 16379
```

Írjunk programot, amely karaktereket olvas a pont karakterig, majd a pont beolvasása után a karaktereket fordított sorrendben kiírja a képernyőre! (*karfordit*)

```
program karfordit;

procedure verem();
var
   kar: char;
begin
   read(kar);
   if kar <> '.' then
     begin
       verem();
       write(kar);
     end;
end;
```

```
begin
  writeln('Kérek egy mondatot ponttal lezárva: ');
  verem();
  readln;
  readln;
end.
```

*A program futásának eredménye:*

```
Kérek egy mondatot ponttal lezárva:
Free Pascal nyelv.
vleyn lacsaP eerF
```

Készítsünk rekurzív függvényt, egy valós szám nem negatív, egész kitevőjű hatványának meghatározására! (*rekhatvany*)

```
program rekhatvany;

function rhatvany(a: real; k: integer):real;
var
  eredmeny: real;
begin
  if k = 0 then
     eredmeny := 1
  else
     eredmeny := a * rhatvany(a,k-1);
  rhatvany := eredmeny;
end;

var
  alap : real;
  kitevo: integer;
begin
  write('alap  : ');
  readln(alap);
  write('kitevő: ');
  readln(kitevo);
  writeln(alap:6:1,'^',kitevo,' = ', rhatvany(alap,kitevo):6:1);
  readln;
end.
```

*A program futásainak eredménye:*

```
alap  : 2
kitevő: 0
   2.0^0 =     1.0

alap  : 2
kitevő: 12
   2.0^12 = 4096.0

alap  : 12
kitevő: 2
  12.0^2 =  144.0
```

Írjunk rekurzív függvényt egy számtani sorozat adott tagjának kiszámítására! (*szamtsor*)

```
program szamtsor;

function an(a1: integer; d: integer; n: integer):longint;
begin
   if n = 1 then
      an := a1
   else
      an := d + an(a1,d,n-1);
end;

var
  elso, diff, szam, n_edik: integer;
begin
  write('első tag       : ');   readln(elso);
  write('különbség      : ');   readln(diff);
  write('hányadik elem : ');   readln(szam);
  n_edik:= an(elso, diff, szam);
  writeln('A sorozat ',szam,'. tagja: ',n_edik);
  readln;
end.
```

*A program futásának eredménye:*

```
első tag       : 2
különbség      : 3
hányadik elem : 7
A sorozat 7. tagja: 20
```

Írjunk rekurzív eljárást, amely a megadott egész szám jegyeit fordított sorrendben írja ki a képernyőre! (*szamforditas*)

```
program szamforditas;

procedure szamfordit(szam: longint);
begin
  write(szam mod 10);
  if szam div 10 <> 0 then szamfordit(szam div 10);
end;

var
  egesz : longint;
begin
  write('Kérek egy egész számot: ');
  readln(egesz);
  write('Az egész szám fordítva: ');
  szamfordit(egesz);
  writeln;
  readln;
end.
```

*A program futásának eredménye:*

```
Kérek egy egész számot: 122379
Az egész szám fordítva: 973221
```

Készítsünk rekurzív és nem rekurzív függvényt a Pascal-háromszög adott sorszámú sorának meghatározására! (*PascalHaromszog*)

Az alábbi ábrán a Pascal-háromszög első hat sora látható. Ha az elemek soron belüli indexe $k$ ($0 \leq k \leq n$), akkor $p(n,k)$ jelölje az $n$. sor, $k$. elemét (például $p(4,3) = 6$).

```
n = 0                          1
n = 1                      1       1
n = 2                  1       2       1
n = 3              1       3       3       1
n = 4          1       4       6       4       1
n = 5      1       5      10      10       5       1
n = 6  1       6      15      20      15       6       1
```

Az elemek meghatározásának rekurzív algoritmusa:
ha $k=0$ vagy $k=n$, akkor $p(n,k)=1$ , egyébként pedig $p(n,k) = p(n-1,k-1) + p(n-1,k)$

A nem rekurzív algoritmus valójában a binomiális együtthatók képlete:

$$p(n,k) = \binom{n}{k} = \frac{n!}{k!\cdot(n-k)!} = \frac{n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot(n-k+1)}{k!}$$

```
program PascalHaromszog;

// rekurzív függvény
function rbinom (n: integer; k: integer) : longint;
var
   binom: integer;
begin
   if (k = 0) or (k = n) then
      rbinom := 1
   else
      rbinom := rbinom(n-1,k-1)+ rbinom(n-1,k);
end;

// nem rekurzív függvény
function binom (n: integer; k: integer): longint;
var
   g: integer;
begin
   binom:= 1;
   for g:=n-k+1 to n do
      binom := binom * g;
   for g:=1 to k do
      binom:= binom div g;
end;
```

```
var
  sorszam,i : integer;
begin
  write('Hányadik sor: ');  readln(sorszam);
  writeln('Nem rekurzív');
  for i:=0 to sorszam do
    write(binom(sorszam,i):4);
  writeln;
  writeln('Rekurzív');
  for i:=0 to sorszam do
    write(rbinom(sorszam,i):4);
  writeln;
  readln;
end.
```

*A program futásainak eredménye:*

```
Hányadik sor: 0
Nem rekurzív
   1
Rekurzív
   1
Hányadik sor: 6
Nem rekurzív
   1   6  15  20  15   6   1
Rekurzív
   1   6  15  20  15   6   1
```

Írjunk rekurzív függvényt a kamatos kamat, valamint az évenkénti összeg kiszámítására! (*kamatrk*)

```
program kamatrk;

function kamat(ev:integer; osszeg,kamatlab:real):real;
begin
   if ev = 0 then
      kamat:=osszeg
   else
      begin
        osszeg := osszeg*(1+kamatlab/100);
        kamat := kamat(ev-1,osszeg,kamatlab);
      end;
end;

procedure kamatkiir(evk,ev:integer; osszeg,kamatlab:real);
begin
   if ev <> 0 then
      begin
         osszeg:=osszeg*(1+kamatlab/100);
         writeln('Az ',evk:2,'. év végén az összeg: ', osszeg:10:2);
         kamatkiir(evk+1,ev-1,osszeg,kamatlab);
      end;
end;
```

```
var ered, osszeg, kamatlab : real;
    ev : integer;
 begin
   writeln('Kérem az összeget és az éves kamatlábat (%-ban)!');
   write('Az összeg      : '); readln(osszeg);
   write('A kamatláb (%): '); readln(kamatlab);
   repeat
     write('Év (1..10)     : ');
     readln(ev);
   until ev in [1..10];
   writeln('Kamatos kamat számítása');
   kamatkiir(1,ev,osszeg,kamatlab);
   ered:=kamat(ev,osszeg,kamatlab);
   writeln;
   writeln('Az összeg: ',ered:10:2);
   readln;
  end.
```

*A program futásának eredménye:*

```
Kérem az összeget és az éves kamatlábat (%-ban)!
Az összeg      : 100000
A kamatláb (%): 10
Év (1..10)     : 10
Kamatos kamat számítása
Az  1. év végén az összeg:  110000.00
Az  2. év végén az összeg:  121000.00
Az  3. év végén az összeg:  133100.00
Az  4. év végén az összeg:  146410.00
Az  5. év végén az összeg:  161051.00
Az  6. év végén az összeg:  177156.10
Az  7. év végén az összeg:  194871.71
Az  8. év végén az összeg:  214358.88
Az  9. év végén az összeg:  235794.77
Az 10. év végén az összeg:  259374.25

Az összeg:  259374.25
```

# 9. Programmodulok

Nagyobb alkalmazások fejlesztését a legtöbb programozási nyelv a forráskód részekre bontásával, és az egyes részek önálló állományban való tárolásával támogatja. Ezek az önállóan lefordítható forrásfájlok a modulok. A moduláris programozás elvének megfelelően a modulok közötti kapcsolat kialakítása során általában az adatrejtés elve is érvényesül.

A könyv eddigi részeiben csak a főprogram (**program**) modult használtuk, melynek deklarációi kívülről (más forrásfájlból) nem érhetők el, így egyedüli szerepe az alkalmazás indítása, futtatása. Míg programmodulból csak egyet tartalmazhat az alkalmazás, addig a további (**unit** és **library**) modulokból tetszőleges számú szerepelhet benne.

Fontos programfejlesztési szempont, hogy a már megírt, jól működő programrészeket ne kelljen újraírni, egy következő feladat során. Hagyományos módon a lefordított eljárásokat és a függvényeket ún. könyvtárakban tárolják, ahonnan kétféleképpen érhetők el.

Statikus szerkesztésről beszélünk, amikor a futtatható program előállítása során, a lefordított alprogramok kódja hozzámásolódik a főprogram kódjához. Így egy adott alprogram kódját minden olyan alkalmazás viszi magával, amelyik hivatkozik rá. Free Pascalban a statikus szerkesztést az önálló programegységek (**unit**ok) használatának lehetősége biztosítja. A hagyományos megoldástól eltérően, itt a kódhozzámásolás nem az alprogramok szintjén, hanem a **unit**ok szintjén valósul meg. További különbség, hogy a **unit** modulokból az eljárások és függvények mellett, típusokat, konstansokat és változókat elérhetünk, sőt gondoskodhatunk a modul használatának előkészítéséről is.

A harmadik Free Pascal modul, a **library** valóban könyvtárként viselkedik, hisz csak alprogramok tárolását és elérését támogatja, azonban a kód hozzámásolása nélkül. A megoldás előnye, hogy ugyanazt a programkódot több alkalmazás egyidejűleg használhatja, méghozzá úgy, hogy a kód egyetlen példányban szerepel a memóriában – dinamikus szerkesztés. A dinamikus szerkesztés elvégzése azonban túlmutat a fejlesztői környezet lehetőségein, hisz a többprogramos operációs rendszer aktív támogatása szükséges hozzá. Könyvünkben csak a ***Windows*** platformon mutatjuk be **library** modulok használatát, ahol dinamikus szerkeszthető könyvtáraknak (*DLL – dynamic link library*) hívják őket.

## 9.1 A unit modulok készítése és felhasználása

A **unit**ok (programegységek) bevezetésével módosul Free Pascal programok eddig megismert szerkezete – a forrásprogram több modulból és egyetlen főprogramból épül fel. A futtatható alkalmazás a lefordított főprogram és a lefordított unitok kódjának egybeépítéséből jön létre.

### 9.1.1 A unitok szerkezete

Az adatrejtés elvének megfelelően minden modul két fő részre tagolható. A modul globális definícióit egy kapcsolódási felületen (**interface**) keresztül érhetjük el, míg a saját (lokális) rész (**implementation**) rejtve marad a modulok és a főprogram elől. A unitok általános felépítése sokban emlékeztet a főprogram szerkezetére:

```
unit modulnév;                  // a unit fejléce
interface                       // a kapcsolódási felület
    uses unitlista;             // a deklarációkhoz használt unitok
    const konstans-definíciók;
    type típus-deklarációk;
    var változó-deklarációk;
    // a globális eljárások és függvények fejlécei

implementation                  // az implementációs rész
    uses unitlista;
    label címkelista;
    const konstans-definíciók;
    type típus-deklarációk;
    var változó-deklarációk;
    // globális eljárások és függvények teljes definíciója
    // lokális eljárások és függvények teljes deklarációja

(initialization                 // az inicializációs rész
    { Pascal utasítások } )
(finalization                   // a lezáró rész
    { Pascal utasítások } )
 end.
```

Megjegyezzük, hogy az inicializációs és a lezáró részek megadása nem kötelező. Amennyiben hiányzik a lezáró rész, akkor az **initialization** helyett **begin** is használható.

#### *A unitok fejléce - hivatkozás a modulokra*

A modulok fejléce a **unit** foglalt szóból egy modulnévből áll. A unitnévnek meg kell egyeznie a modult tároló állomány nevével. Például a

```
unit Transzformaciok;
```

fejlécű modult a *Transzformaciok.pas* állományban kell elhelyeznünk.

A unit nevét más modulokkal, illetve a főprogrammal való kapcsolat kialakítására használjuk:

```
uses modullista;
```

A **uses** („használja") foglalt szót az elérni kívánt modulok vesszővel tagolt listája követi, például:

```
uses Sysutils, Transzformaciok;
```

Az **interface** részben megadott globális deklarációkra ugyanúgy hivatkozhatunk, mint a program szintjén definiált típusokra, konstansokra, változókra és alprogramokra. Ha azonban egy globális azonosító a főprogram által használt több unitban is előfordul, akkor a modulnév megadásával a hivatkozást egyértelművé tehetjük, mint az alábbi, az aktuális dátumot és időt megjelenítő utasításban:

```
writeln(SysUtils.DateTimeToStr(SysUtils.Date+SysUtils.Time));
```

Alapértelmezés szerint a modulokat a fordító az aktuális könyvtárban keresi, azonban a **uses** sorban azt is megadhatjuk, hogy mely állományok tárolják a felhasználni kívánt unitokat:

```
uses SysUtils, Modul1 in 'Modul1.pas',
               Modul2 in '..\Modulok\Modul2.PAS',
               Modul3 in 'Modul3.ppu';
```

### *Az interface rész*

A kapcsolódási felület részt az **interface** foglalt szó vezeti be. Ebben a részben találhatók a unit globális deklarációi (típusok, konstansok, változók és alprogramok), amelyek más modulokból és a főprogramból is elérhetők. Az interfész részben a globális eljárásoknak és függvényeknek csak a fejlécét adjuk meg (az alprogramok teljes definícióját a unit **implementation** rész tartalmazza).

A **const**, a **type**, a **var**, a **procedure** és a **function** deklarációkat tetszőleges számban és sorrendben elhelyezhetjük. Ha az **interface** részben egy másik modulban definiált típusra kívánunk hivatkozni, akkor a hivatkozott modul nevét a **uses** programsorban kell megadnunk.

### *Az implementation rész*

A unitok implementációs része egyrészt az **interface** részben megadott, globális eljárások és függvények teljes definícióját, másrészt pedig a modul lokális deklarációit (címkék, konstansok, típusok, változók, eljárások és függvények) tartalmazza. A globális alprogramok fejlécét az **implementation** részben ajánlott egy az egyben megismételni, mivel ellenkező esetben a túlterhelés mechanizmusa érvényesül.

Az implementációs részben deklarált típusos konstansok valamint az alprogramokon kívül definiált változók, a globális definíciókhoz hasonlóan, statikus élettartamúak, azonban az elérhetőségük az implementációs részre korlátozódik.

*Az inicializációs és a lezáró rész*

Az **initialization** részben a unitot inicializáló programutasítások találhatók. Az inicializációs rész módot ad a modul működéséhez szükséges előkészítő tevékenységek elvégzésére (például kezdőértékek beállítása). A modulok inicializációs része a program indításakor, a főprogram törzse előtt hajtódik végre. A modul inicializációja csak egyszer történik meg, függetlenül attól, hogy hányszor szerepel a modul neve a különböző **uses** hivakozásokban. Ha nincs lezáró rész, akkor az **initialization** helyett használhatjuk a **begin** kulcsszót is.

A **finalization** részben a főprogram törzséből való kilépés utáni, modullal kapcsolatos teendőkről gondoskodhatunk.

Mindkét rész elhagyható, azonban szükség esetén az inicializáló részt önállóan is használhatjuk. Amennyiben az inicializációs és lezáró részt is elhagyjuk, akkor az **end**. utasítás az implementációs részt zárja.

### 9.1.2 A Free Pascal unitjai

A Free Pascal moduljainak összességét *Run Time Library*-nek (*RTL*) nevezzük. A unitok közül a ***System*** modul minden programhoz automatikusan hozzákapcsolódik. ***ObjFPC*** és ***Delphi*** módokban a ***System*** unit szerepét az ***ObjPas*** modul veszi át.

| | |
|---|---|
| ***BaseUnix*** | Alapvető Unix alprogramok és típusok |
| ***Classes*** | Az RTL és a *Free Component Library* (*FCL*) alaposztályai (***TStream***, ***TFileStream***, ***TList***, ***TStringList***, ***TCollection*** stb.) és a kapcsolódó kivételosztályok, alprogramok. |
| ***Crt*** | Turbo Pascal képernyő- és billentyűzetkezelő unit. |
| ***dateutils*** | A *TDateTime* típushoz kapcsolódó dátum- és időkezelő alprogramok. |
| ***Dos*** | Turbo Pascal MS-DOS programozási felület. |
| ***dxeload*** | Betölti a DXE állományt a memóriába. |
| ***dynlibs*** | A dinamikusan betöltődő könyvtárak használatának keresztplatformos támogatása. |
| ***emu387*** | Aktivizálja a matematikai társprocesszor emulációját. |
| ***getopts*** | GNU-kompatibilis elérése a parancssorban megadott opcióknak. |
| ***go32*** | A 32-bites DOS *extender* (DOS-bővítő) elérését segítő modul. |
| ***gpm*** | Interfész a GPM (*General Purpose Mouse* - általános célú egérkezelő) könyvtárhoz. |

| | |
|---|---|
| ***Graph*** | A képernyőgrafika kezelését támogató Turbo Pascal kompatibilis unit. |
| ***heaptrc*** | A halomterületen (*heap*) végzett nyomkövetést segítő modul. |
| ***ipc*** | A Unix alatti, folyamatok közötti kommunikáció (*Inter Process Communication*) alprogramjai, típusai és konstansai. |
| ***Keyboard*** | Az alacsony szintű billentyűzet-funkciók elérését segítő alprogramok, típusok és konstansok. |
| ***Linux*** | A Linux-specifikus, operációs rendszer szintű hívások modulja. |
| ***math*** | Kibővített matematikai könyvtár. |
| ***mmx*** | Az MMX processzorfunkciók elérését segítő unit. |
| ***Mouse*** | Az egéreseményekkezelését segítő modul. |
| ***Objects*** | Turbo Pascal kompatibilis alapobjektumok unitja. (***TStream***, ***TStringList***, ***TCollection***, ***TPoint*** stb. - a *Free Vision* könyvtár objektumtípusai.) |
| ***objpas*** | ***ObjFPC*** és ***Delphi*** módokban a ***System*** unit szerepét átvevő unit. |
| ***oldlinux*** | Elavult Linux modul - csak a kompatibilitás miatt maradt meg a Free Pascalban. |
| ***ports*** | A hardverportok elérését támogató modul. |
| ***printer*** | A nyomtató használatát segítő unit. |
| ***Sockets*** | TCP/IP socketeken keresztüli kommunikációt támogató modul. |
| ***strings*** | Nullavégű sztringek (***pchar***) használatát segítő alprogramok unitja. |
| ***strutils*** | Különböző karaktersorozat-kezelő függvények és eljárások modulja. |
| ***System*** | A rendszerunit. |
| ***sysutils*** | A rendszermodult bővítő típusok, konstansok, eljárások és függvények unitja. |
| ***typinfo*** | A futás közbeni típusinformációk (*Run-Time Type Information - RTTI*) feldolgozását segítő unit. |
| ***Unix*** | Bővített Unix alprogramok és típusok. |
| ***unixtype*** | Unix alaptípusok. |
| ***unixutil*** | Unix segédprogramok modulja. |
| ***Video*** | Képernyőkezelő modul. |
| ***x86*** | A Unix platformokhoz készült, Intel X86-specifikus rutinok unitja. |

### 9.1.3 Példaprogramok a unit modulok alkalmazására

A saját unitok tervezése és készítése fontos részét képezi a programfejlesztői munkának. Jól kialakított modulok meggyorsíthatják, és biztonságosabbá tehetik a későbbi programozási feladatok végzését. A következőkben három példán keresztül mutatjuk be a unitok készítésének és használatának fogásait.

Tervezzünk **unit** modult a síkbeli geometriai műveletek végzésére! Készítsünk programmodult, a unit használatának bemutatására! (*\Geom2DUnit)*

A modult *Geom2DUnit* néven a *Geom2DUni.pas* forrásállományban tároljuk. A **unit** önállóan lefordítható, aminek eredménye a *Geom2DUnit.ppu* állomány. A műveleteket végző alprogramok mellett típust, konstanst és globális változót is elhelyeztünk a modulban, valamint használtuk az inicializációs részt.

```
unit Geom2DUtils;

// Az interfész rész
interface

type
     TPont = record
         x, y : real;
     end;

const
     Origo : TPont = (x:0; y:0);

var
     XTengely : array[1..2] of TPont;
     YTengely : array[1..2] of TPont;

procedure beallit(var p : TPont; x,y : real);
function tavolsag(const p, q : TPont) : real;
procedure eltol(var p : TPont; dx,dy : real);
procedure elforgat(var p : TPont; beta : real);
function pstr(const p : TPont) : string;

// Az implementációs rész
implementation
uses sysutils;

procedure beallit(var p : TPont; x,y : real);
begin
  p.x := x;
  p.y := y;
end;

function tavolsag(const p, q : TPont) : real;
begin
   tavolsag := sqrt(sqr(p.x-q.x)+sqr(p.y+q.y));
end;

procedure eltol(var p : TPont; dx,dy : real);
begin
  p.x := p.x + dx;
  p.y := p.y + dy;
end;
```

```
procedure elforgat(var p : TPont; beta : real);
var
  alfa, tav  : real;
begin
  alfa := arctan(p.y/p.x);
  tav  := tavolsag(p,Origo);
  p.x := tav*cos(alfa+beta);
  p.y := tav*sin(alfa+beta);
end;

function pstr(const p : TPont) : string;
begin
  pstr :='('+floattostrF(p.x, ffFixed,8,3)+' , '
           +floattostrF(p.y,ffFixed,8,3)+')';
end;

// Az inicializációs rész
initialization
  XTengely[1].x:=-1;     XTengely[1].y:= 0;
  XTengely[2].x:=+1;     XTengely[2].y:= 0;
  YTengely[1].x:= 0;     YTengely[1].y:=-1;
  YTengely[2].x:= 0;     YTengely[2].y:=+1;

// A lezáró rész - nincs semmi teendőnk
finalization
end.
```

A *Geom2DUtils* modul működését bemutató főprogramot a program modul tartalmazza a *FoProg.pas* állományban:

```
program geom2d;

uses Geom2DUtils;

var
   p1, p2, p3 : TPont;
begin
   beallit(p1,1,1);
   writeln(pstr(p1));
   eltol(p1,3,4);
   writeln(pstr(p1));
   elforgat(p1,pi);
   writeln(pstr(p1));
   p2 := XTengely[2];
   p3 := YTengely[2];
   writeln(Pstr(p2),' és ', Pstr(p3) ,
           ' pontok távolsága: ', tavolsag(p2,p3):6:4);
   readln;
end.
```

*A program futásának eredménye:*

```
(1,000 , 1,000)
(4,000 , 5,000)
(-4,000 , -5,000)
(1,000 , 0,000) és (0,000 , 1,000) pontok távolsága: 1.4142
```

Tervezzünk unitot, amely segíti a téglalapok adatainak tárolását és számítását! Írjunk főprogramot, a modul használatának bemutatására! (*\TeglalapUnit)*

A téglalap oldalhosszait a *Teglalap* rekordban tároljuk, míg a hosszak beolvasását és a számításokat globális eljárások végzik. A modult a *TeglalapU.pas* állomány tartalmazza.

```
unit TeglalapU;

Interface

type
    Teglalap = record
       // adatmezők
       a, b  : integer;
    end;

procedure Olvas(var t:Teglalap);
procedure TerKerSzamol(t : Teglalap; var ter, ker:real);

Implementation

procedure Olvas(var t:Teglalap);
begin
      writeln('Téglalap oldalai');
      write('A oldal: '); readln(t.a);
      write('B oldal: '); readln(t.b);
end;

procedure TerKerSzamol(t : Teglalap; var ter, ker:real);
begin
      ter := t.a * t.b;
      ker := (t.a + t.b)*2;
end;

end.
```

A *TeglalapU* unitot a *TeglalapPr.pas* állományban tárolt programmodulból használjuk:

```
program TeglalapPr;
uses Teglalapu;
var
   rt : Teglalap;
   t_ter, t_ker: real;
begin
   Olvas(rt);
   TerKerSzamol(rt,t_ter,t_ker);
   writeln('A téglalap területe : ',t_ter:6:1);
   writeln('A téglalap kerülete : ',t_ker:6:1);
   readln;
end.
```

*A program futásának eredménye:*

```
Téglalap oldalai
A oldal: 2
B oldal: 3
A téglalap területe :    6.0
A téglalap kerülete :   10.0
```

Készítsünk diákok adatait nyilvántartó programot! A szükséges típusdefiníciókat, valamint a beolvasó, a legnagyobb átlagot és a legkisebb kreditpontot kereső alprogramokat helyezzük unit modulba! Írjunk főprogramot, amely a modult használva megjeleníti az eredményeket! (*\DiakUnit*)

A *DiakU* modult a *DiakU.pas* állomány tárolja:

```
unit DiakU;

Interface

type
    diak = record
        neve: string;
        tk  : integer;
        kredit: integer;
        atlag: real;
    end;
    tankor= array[1..30] of diak;

procedure MaxAtlag(n:integer; t:tankor; var max_atlag:real);
function  MinKredit(n:integer; t:tankor):integer;
procedure Olvas(var n: integer; var t:tankor);

Implementation

procedure MaxAtlag(n:integer; t:tankor; var max_atlag:real);
var i: integer;
begin
    max_atlag:=t[1].atlag;
    for i:=2 to n do
       if max_atlag<t[i].atlag then
           max_atlag:= t[i].atlag;
end;

function MinKredit (n:integer; t:tankor):integer;
var i:integer;
begin
    MinKredit:=t[1].kredit;
    for i:=2 to n do
       if MinKredit > t[i].kredit then
           MinKredit:= t[i].kredit;
end;
```

```
procedure Olvas(var n: integer; var t:tankor);
var i: integer;
begin
    write('A diákok száma: '); readln(n);
    for i:=1 to n do
      begin
        writeln; writeln(i:2,'. diak: ');
        write('Neve  : '); readln(t[i].neve);
        write('Tankör: '); readln(t[i].tk);
        write('Kredit: '); readln(t[i].kredit);
        write('Átlag: '); readln(t[i].atlag);
      end;
end;
end.
```

A tesztelő főprogramot a *DiakPr.pas* állományban találjuk:

```
program DiakPr;
uses DiakU;
var
  t1: tankor;
  db: integer;
  mk1: real;
  mk2: integer;

begin
  Olvas(db,t1);
  MaxAtlag(db,t1,mk1); writeln;
  writeln('Legjobb átlag   : ',mk1:6:2);
  mk2:=MinKredit(db,t1);
  writeln('Legkisebb kredit: ',mk2);
  readln;
end.
```

*A program futásának eredménye:*

```
A diákok száma: 3

 1. diak:
Neve  : Kiss Antal
Tankör: 1
Kredit: 45
Átlag: 3

 2. diak:
Neve  : Nagy István
Tankör: 1
Kredit: 78
Átlag: 4

 3. diak:
Neve  : Komáromi Anna
Tankör: 1
Kredit: 89
Átlag: 5

Legjobb átlag   :   5.00
Legkisebb kredit: 45
```

## 9.2 A library modulok használata

A dinamikusan szerkeszthető/betölthető könyvtárak (*DLL*) készítésével lehetőség nyílik arra, hogy több alkalmazás megosztva használjon programkódot. A Free Pascal a **library** modullal támogatja a dinamikusan szerkeszthető könyvtárak kialakítását.

A fejezet első részében ismertetett Pascal modulok (unitok) sok szempontból különböznek a könyvtáraktól:

- A könyvtárakból csak függvényekre és eljárásokra hivatkozhatunk. Típusok, konstansok és változók elérése nem lehetséges.
- A lefordított **unit**ot **.ppu** kiterjesztésű állományban tárolja a rendszer. Ha azonban a **unit** forrásnyelven megtalálható, akkor az főprogram fordításakor (*Compile/Build*) a Free Pascal lefordítja azt. Ezzel szemben a **library** és az őt használó főprogram fordítása független egymástól. A lefordított könyvtár Windows alatt **.dll** kiterjesztésű állományban jön létre.
- A Free Pascal nyelven megírt könyvtárakat más nyelven (C, C++ stb.) készült alkalmazások is használhatják. Ehhez azonban módosítókkal be kell állítanunk a megfelelő hívási konvenciókat: ***pascal***, ***cdecl***, ***stdcall***, ***safecall***, ***varargs***.

### 9.2.1 Dinamikusan szerkeszthető/betölthető könyvtárak készítése

A könyvtármodul szerkezete leginkább a főprogram felépítésére hasonlít – ugyanúgy megtaláljuk benne a deklarációs és a futtatható részt. A **library** azonban a deklarációs részen belül további szekciókat is tartalmazhat (**exports**), ahol opcionális névvel (***name***) ellátva, vesszővel elválasztva felsorolhatjuk azokat a rutinokat, amelyeket kívülről is hívhatók (amelyeket „exportálunk"). Az **exports** részek tetszőleges számú elemet tartalmazhatnak, és a deklarációs részen belül bárhova elhelyezhetők.

```
library könyvtárnév;          // a library fejléce
    uses unitlista;           // a felhasznált unitok
    label címkelista;
    const konstans-definíciók;
    type típus-deklarációk;
    var változó-deklarációk;
    // eljárások és függvények teljes definíciója
    exports alprogramnév-lista;

begin
    // A könyvtárat inicializáló Pascal utasítások ]
end.
```

A könyvtárat tetszőleges Pascal forrásállományban tárolhatjuk, függetlenül a fejlécben megadott névtől. A lefordított könyvtár felveszi a forrásfájl nevét, a kiterjesztése Windows alatt *.dll* lesz.

Az **exports** részben megadott alprogramokat más programból a saját nevükkel, vagy a ***name*** direktíva után álló névvel érhetjük el (a nevek kis- és nagybetű érzékenyek). A Windows minden névhez egy egész sorszámot rendel, melyet magunk is beállíthatunk az ***index*** direktívát követően.

```
exports
  // exportálás saját névvel:
  eljaras1, fuggveny1,
  // exportálás indexszel:
  eljaras2 index 12,
  // exportálás új névvel és indexszel:
  fuggveny2 index 23 name 'Fuggveny2',
  // exportálás új névvel:
  eljaras3 name 'Eljaras3';
```

A könyvtármodult záró utasításblokk a **library** globális változóit inicializálja. A dinamikusan szerkeszthető könyvtár globális adatainak másolatát minden olyan alkalmazás megkapja, amely a könyvtárat használja. A könyvtár inicializációs része minden új hivatkozó alkalmazás indításakor végrehajtódik.

### 9.2.2 Dinamikusan szerkeszthető/betölthető könyvtárak felhasználása

Ahhoz hogy a könyvtárban definiált alprogramok elérhetők legyenek *„importálnunk"* kell őket. Minden exportált könyvtárrutin külső definíció a programunk számára (***external***). Az importálást az **exports** szekcióban megadottaknak megfelelően háromféleképpen végezhetjük – az alprogram neve, indexe valamint a beállított neve alapján:

```
procedure eljaras1; external 'DLLnev';
procedure elj2; external 'DLLnev' index 12;
procedure elj3; external 'DLLnev' name 'Eljaras3';
function fuggveny1:integer; external 'DLLnev';
function fv2:integer; external 'DLLnev' index 23 ;
function fv22:integer; external 'DLLnev' name 'Fuggveny2';
```

Az elmondottak alapján tetszőleges DLL-ben tárolt alprogramot meghívhatunk. Az importáláshoz azonban pontosan kell ismernünk az eljárás vagy függvény nevét (vagy indexét), illetve a paraméterezését. Hasznos megoldás, ha a mindenkori importálás helyett, ún. ***importunitot*** hozunk létre, amelyben akár több DLL importálását is elvégezhetjük.

```
unit ImportUnit;
interface
   procedure eljaras1;
   procedure elj2;
   procedure elj3;
   function fuggveny1:integer;
   function fv2:integer;
```

```
    function fv22:integer;

implementation
    procedure eljaras1; external 'DLLnev';
    procedure elj2; external 'DLLnev' index 12;
    procedure elj3; external 'DLLnev' name 'Eljaras3';
    function fuggveny1:integer; external 'DLLnev';
    function fv2:integer; external 'DLLnev' index 23 ;
    function fv22:integer; external 'DLLnev' name 'Fuggveny2';
end.
```

Az **interface** részben csak az alprogramok fejlécét adjuk meg. Az importálás az **implementation** részben történik.

### 9.2.3 Példaprogram a könyvtármodul kialakítására

Az előző rész síkbeli geometriai példájában szereplő eljárásokat és függvényeket helyezzük el **library** modulban (*Geom2DLib.pas*)! Készítsünk a könyvtárhoz importunitot (*Geom2DImport.pas*), és főprogramot (*Foprog.pas*)! Az alkalmazás állományait tároljuk a *\Geom2DLibrary* alkönyvtárban!

A *Geom2DLib.pas* fájl fordításával az aktuális könyvtárban létrejön a *Geom2DLib.dll* modul.

```
library Geom2DLib;
uses SysUtils;

type
    TPont = record
        x, y : real;
    end;

const
    Origo : TPont = (x:0; y:0);

procedure beallit(var p : TPont; x,y : real);
begin
  p.x := x;
  p.y := y;
end;

function tavolsag(const p, q : TPont) : real;
begin
   tavolsag := sqrt(sqr(p.x-q.x)+sqr(p.y+q.y));
end;

procedure elforgat(var p : TPont; beta : real);
var
  alfa, tav  : real;
begin
  alfa := arctan(p.y/p.x);
  tav  := tavolsag(p,Origo);
  p.x := tav*cos(alfa+beta);
```

```
  p.y := tav*sin(alfa+beta);
end;
procedure eltol(var p : TPont; dx,dy : real);
begin
  p.x := p.x + dx;
  p.y := p.y + dy;
end;

function pstr(const p : TPont) : string;
begin
  pstr :='('+floattostrF(p.x, ffFixed,8,3)+' , '
           +floattostrF(p.y,ffFixed,8,3)+')';
end;

// Az exportálandó alprogramok listája
exports
  beallit, tavolsag,
  eltol, elforgat, pstr;

// A DLL inicializálása
begin
  // nincs semmi teendőnk
end.
```

Az importunitban meg kell ismételnünk a típus- és konstans-definíciókat.

```
unit Geom2DImport;

interface
type
     TPont = record
         x, y : real;
     end;

const
     Origo : TPont = (x:0; y:0);

var
     XTengely : array[1..2] of TPont;
     YTengely : array[1..2] of TPont;

procedure beallit(var p : TPont; x,y : real); external 'Geom2DLib';
function tavolsag(const p, q : TPont) : real; external 'Geom2DLib';
procedure eltol(var p : TPont; dx,dy : real); external 'Geom2DLib';
procedure elforgat(var p : TPont; beta : real);external 'Geom2DLib';
function pstr(const p : TPont) : string; external 'Geom2DLib';

implementation

initialization
  XTengely[1].x:=-1;     XTengely[1].y:= 0;
  XTengely[2].x:=+1;     XTengely[2].y:= 0;
  YTengely[1].x:= 0;     YTengely[1].y:=-1;
  YTengely[2].x:= 0;     YTengely[2].y:=+1;
```

```
finalization
  // Nincs semmi dolgunk
end.
```

A főprogram csak a **uses** sorban különbözik a korábban közölt változattól:

```
program geom2d;

uses Geom2DImport;

var
   p1, p2, p3 : TPont;

begin
   beallit(p1,1,1);
   writeln(pstr(p1));
   eltol(p1,3,4);
   writeln(pstr(p1));
   elforgat(p1,pi);
   writeln(pstr(p1));
   p2 := XTengely[2];
   p3 := YTengely[2];
   writeln(Pstr(p2),' és ', Pstr(p3) ,
           ' pontok távolsága: ', tavolsag(p2,p3):6:4);
   readln;
end.
```

*A program futásának eredménye:*

```
(1,000 , 1,000)
(4,000 , 5,000)
(-4,000 , -5,000)
(1,000 , 0,000) és (0,000 , 1,000) pontok távolsága: 1.4142
```

Megjegyezzük, hogy a *Geom2DLib* könyvtárat közvetlenül, az eredeti példában szereplő *Geom2DUtils* unit felhasználásával is előállíthattuk volna. Ezt a megoldást a \Geom2DLibrary2 alkönyvtár tartalmazza.

A *Geom2DLib.pas* állomány módosított változata:

```
library Geom2DLib;
uses SysUtils, Geom2DUtils;

// Az exportálandó alprogramok listája
exports
  beallit, tavolsag,
  eltol, elforgat, pstr;

// A DLL inicializálása
begin
  // nincs semmi teendőnk
end.
```

# 10. Dinamikus memóriahasználat

Az eddig használt egyszerű és strukturált típusú változók közös tulajdonsága, hogy rájuk a deklarálás során megadott névvel hivatkozunk. Az így definiált változók mérete a program fordításakor dől el, és a program futása során nem módosítható. A Free Pascal nyelvben ezeket a változókat statikus helyfoglalású változóknak nevezzük.

A hatékony memória-felhasználás érdekében szükség van arra, hogy bizonyos memóriaterülettel a programozó szabadon gazdálkodjon. A Free Pascal nyelv rendelkezik olyan eszközökkel, amelyek segítségével a programozó maga foglalhat memóriaterületet a változói számára. A lefoglalt területet fel is szabadíthatja, ha már nincs szüksége a változóra. A felszabadított terület egy következő memóriafoglalás során újra felhasználható. A memóriafoglalás és felszabadítás műveleteivel a memória dinamikus felhasználását valósítjuk meg. A következőkben megismerkedünk a dinamikus memóriakezelés eszköztárával.

## 10.1 A mutatótípus és a dinamikus változók

Mint ahogy azt az előző fejezetekben láttuk, a

```
var a : integer;
```

deklaráció hatására a fordítóprogram egy ***sizeof(integer)*** méretű területet foglal le a memóriában, az *a* változó számára. Az *a* változóra nevének felhasználásával hivatkozhatunk:

```
a:= 23;
a:= a+12;
```

Ahhoz, hogy a memóriafoglalás fenti lépéseit a program futása során végezzük el, meg kell ismerkednünk egy új változótípussal, a mutatóval (*pointerrel*). A mutató olyan speciális változó, amely memóriacímet tárol. A Pascal nyelvben a mutatók lényeges tulajdonsága, hogy típussal rendelkeznek. A típus kijelöli, hogy a dinamikus helyfoglalás során mekkora területet kell a mutatóhoz hozzárendelni. A mutatók deklarálásának általános formája:

```
var változónév : ^típusnév;
```

ahol a típusnév tetszőleges szabványos, vagy felhasználó által definiált típus neve. Például:

```
var ap : ^integer;
```

Az *ap* mutató a deklarálás után semmilyen meghatározott címet sem tartalmaz, tehát még „sehova se mutat".

Free Pascal-ban a mutatónak többféleképpen is lehet értéket adni. Most csak a legegyszerűbb szabványos megoldással foglalkozunk. A programon belüli helyfoglalás elvégzésére a ***new()*** eljárást használjuk, amely a lefoglalt terület címét a paraméterként átadott mutatóba tölti:

```
New(ap);
```

A Free Pascal-ban létezik az előre definiált **nil** konstans, amelyet a nulla (üres) mutató jelölésére használunk:

```
ap:= nil;
```

Ha a memóriafoglalás sikeres volt, következhet a lefoglalt terület felhasználása. A mutató által kijelölt memóriaterületre a mutató nevével, és az utána megadott ^ jellel együtt hivatkozunk:

```
ap^:= 23;
ap^:= ap^+12;
```

Jól látható a fejezet elején definiált *a* változó és az *ap^* hivatkozás közötti analógia. A mutató neve után álló ^ jel hatására nem a mutatót, hanem az általa kijelölt memóriaterületet érjük el.

Ha már nincs szükségünk az *ap^* egész típusú változóra (a dinamikusan lefoglalt területre), akkor azt a ***Dispose()*** eljárás hívásával törölhetjük:

```
Dispose(ap);
```

A ***Dispose()*** hívás utáni, illetve a ***New()*** hívás előtti *ap^* hivatkozás programhibához vezet.

A dinamikus helyfoglalás előnye természetesen akkor tűnik ki igazán, ha nem néhánybájtos, hanem több kilobájtos dinamikus változó számára foglalunk tárterületet (például tömböt, rekordot) használunk.

A strukturált típusokhoz csak akkor definiálhatunk mutatót, ha a típushoz egyetlen típusnevet rendelünk, a deklarációs rész **type** szekciójában. Nézzünk néhány mutatódefiníciót és néhány indirekt (közvetett, mutatóval kijelölt) memóriahivatkozást:

```
// Típusdefiníciók
type
   trtart  = 0..1023;
   tvalasz = (igen, nem);
   tsor    = string[80];
```

```
    thalmaz = set of char;
    ttomb   = array [-5..5] of integer;
    trekord = record
                nev : string;
                kor : integer;
              end;

// Mutató-deklarációk
var
    prtart  : ^trtart;
    pvalasz : ^tvalasz;
    psor    : ^tsor;
    phalmaz : ^thalmaz;
    ptomb   : ^ttomb ;
    prekord : ^trekord;
    pfile   : ^file;
    i : integer;

begin
    // helyfoglalás:
    new(prtart);
    new(pvalasz);
    new(psor);
    new(phalmaz);
    new(ptomb);
    new(prekord);
    new(pfile);

    // a lefoglalt területek elérése:

    // a memória elérése egyszerű adattípusokkal:
    prtart^:= 512;
    pvalasz^:= igen;
    psor^:= 'Hello';

   // a memória elérése strukturált adattípusokkal:
    phalmaz^:= ['A'..'Z', 'a'..'z'];

    for i:=-5 to 5 do
      ptomb^[i]:= sqr(i);

    prekord^.nev:= 'Blaise Pascal';
    prekord^.kor:= 362;
    with prekord^ do
     begin
       nev:= 'Blaise Pascal';
       kor:= 362;
     end;
    // a lefoglalt területek felszabadítása:
    dispose(prtart);
    dispose(pvalasz);
    dispose(psor);
    dispose(phalmaz);
    dispose(ptomb);
    dispose(prekord);
end.
```

## 10.2 További memóriakezelő függvények és eljárások

A Free Pascal-ban a ***pointer*** típussal ún. általános mutatót deklarálhatunk. Az így létrehozott mutató nem rendelkezik konkrét típussal, ezért a ***new()*** eljárást nem használhatjuk a memóriafoglalás elvégzésére. Ebben az esetben az adott bájtméretű memóriaterület lefoglalását a ***GetMem()*** vagy az ***AllocMem()***, felszabadítását pedig a ***FreeMem()*** alprogramok (***System*** unit) segítségével végezhetjük el. A memóriaméretet a ***PtrInt*** típussal azonosítja a Free Pascal, amely a mutatók méretével megegyező méretű egész típust jelöli:

```
procedure GetMem(var p: pointer; meret : PtrInt);
function GetMem(meret: PtrInt): pointer;
function AllocMem(meret: PtrInt): pointer;
procedure FreeMem(p: pointer; meret: PtrInt);
function FreeMem(p: pointer): PtrInt;
```

A fenti alprogramok segítségével lefoglalt memóriablokk mérete a ***ReallocMem()*** eljárással módosítható.

```
function ReallocMem(var p: pointer; ujmeret: PtrInt):pointer;
function ReallocMemory(var p: pointer; ujmeret: PtrInt):pointer;
```

Megjegyezzük, hogy a több platform támogatása következtében a Free Pascal moduljaiban bizonyos alprogramok más alprogramok aliasaként szerepelnek.

A pointer típusú mutatókat legtöbbször dinamikus pufferterületek kijelölésére használjuk, melyek méretét a program futása során adjuk meg. Az alábbi példaprogramban a bemutatott eljárások használatát láthatjuk, egy dinamikus helyfoglalású tömb kezelése során:

```
{$R-}
program Get_Free;
const
  maxelemszam=10*1024*1024;
  elemszam1=1024*1024;
  elemszam2=8*1024;
type
  ttomb=array[1..1] of double; // a lefoglalt terület eléréséhez
var
  pt : pointer;
  i  : longint;
begin
  // 8 megabájt memória lefoglalása, és feltöltése
  pt := GetMem(sizeof(double)* elemszam1);
  for i:=1 to elemszam1 do
     ttomb(pt^)[i]:=2*i+1;
```

```
  writeln(ttomb(pt^)[2006]:10:0);
  writeln(ttomb(pt^)[elemszam1]:10:0);

  // a memóriaterület méretének csökkentése 64 kilobájtra
  ReallocMem(pt,elemszam2*sizeof(double));
  writeln(ttomb(pt^)[2006]:10:0);
  writeln(ttomb(pt^)[elemszam2]:10:0);

  // a memóriaterület felszabadítása
  FreeMem(pt);
  readln;
end.
```

## 10.3 Műveletek mutatókkal

Az alábbi alfejezetben áttekintjük azokat a lehetőségeket, amelyek ismeretében avatott felhasználói lehetünk a mutatóknak. Felhívjuk azonban a figyelmet arra, hogy a mutatókkal mindig körültekintően kell bánni. Mint ahogy láttuk, a Free Pascal-ban egyaránt definiálhatunk típusos és típus nélküli mutatókat:

```
var ip : ^byte;      // típusos mutató
    p : pointer;     // típus nélküli mutató
```

Mindkét mutatótípusra értelmezettek az alábbi műveletek:

– ***Értékadás:***

```
ip^:= 4;             // értéket adunk a "mutatott" objektumnak
p  := nil;           // értéket adunk magának a mutatónak
ip := p;
```

– ***Mutatók léptetése:***

Az ***Inc()*** és ***Dec()*** eljárások segítségével a típusos mutatókat előre-, illetve visszaléptethetjük. A léptetés során a mutató értéke annyi bájttal módosul, amennyi a típusának megfelelő adatméret:

```
var
  a : array[1..5] of integer=(2,4,6,8,10);
  p :^integer;
begin
  p:=@a;             // vagy p:=@a[1];
  p^:=0;
  inc(p,3);
  p^:=23;
  dec(p);
  p^:=-12;
  // A tömb elemei: 0, 4, -12, 23, 10
end.
```

– ***Összehasonlítás:***

Azonos típusú mutatók esetén használhatjuk az = és a <> műveleteket. A **nil** konstansmutató és a ***pointer*** típussal deklarált mutatók bármely más mutatóval összehasonlíthatók.

```
if (ip=p) and (ip<>nil) then halt;
```

– ***Típuskonverzió:***

A mutató típusa típuskonverzióval megváltoztatható. Ily módon a pointer típusú mutatókhoz is rendelhetünk konkrét típust. Az alábbi példában az *arr* típusú mutató által kijelölt területre, mint ***longint*** változóra hivatkozunk:

```
type arr = array[1..4] of byte;
var  ap : ^arr;
begin
  new(ap);
  longint(ap^):= $12345678;
  // ez megfelel a következő értékadásoknak I386 környezetben:
   ap^[1]:= $78; ap^[2]:= $56; ap^[3]:= $34; ap^[4]:= $12;
  dispose(ap);
end.
```

– ***Az Addr() függvény és a @ operátor:***

Az *Addr()* függvény, illetve az ennek megfelelő @ operátor segítségével bármely Pascal objektum (függvény, változó stb.) címét lekérdezhetjük, és tetszőleges típusú mutatóba tölthetjük:

```
var ip : ^integer;
     i : integer;

begin
  i  := 5;
  ip := @i;    // vagy ip:= Addr(i);
  ip^:= i + 7;
  writeln(i); // a program által kiírt érték: 12
end.
```

– ***Az Assigned() függvény:***

A könyvtári függvények egy része **nil** értékű mutatóval tér vissza, ha működése során hiba lépett fel. Ezeknél a függvényeknél a visszaadott értéket mindig tesztelnünk kell:

```
if p=nil then
  begin
    // sikertelen művelet
  end;
```

A Free Pascal-ban a fenti ellenőrzés olvashatóbbá tehető az *Assigned()* függvény alkalmazásával:

```
if not Assigned(p) then
   begin
     // sikertelen művelet
   end;
```

A függvény paramétereként valamilyen mutató, vagy eljárás-, metódus- vagy osztálytípusú változó állhat, melyek nem **nil** értéke esetén **true** értékkel tér vissza.

– *Az absolute deklaráció:*

Az **absolute** deklaráció segítségével egy új változót egy már meglévő változó területén hozhatunk létre. Ezzel a megoldással – bizonyos esetekben – elkerülhetjük a mutatók elkalmazását. A fenti, mutatók típuskonverzióját bemutató rész példáját absolute deklarációval is megvalósíthatjuk:

```
var t : array[1..4] of byte;
    a : longint absolute t;
begin
  a := $12345678;
  // ez megfelel a következő értékadásoknak I386 környezetben:
  t[1]:= $78; t[2]:= $56; t[3]:= $34; t[4]:= $12;
end.
```

## 10.4 Dinamikus tömbök használata

A mutatók segítségével valóban hatékonyan használhatjuk számítógépünk memóriáját, a mutatók helytelen használata azonban egy sor programhibához vezethet. Az újabb programnyelvek (Visual Basic, Java) éppen a potenciális hibaforrás, illetve az ellenőrizetlen memóriahozzáférés kizárásának érdekében nem támogatják a mutatók alkalmazását. A memória dinamikus elérésére azonban tartalmaznak megoldást, az ún. dinamikus tömbök programozásával.

A dinamikus tömb méretét – ellentétben a már ismertetett statikus tömbökkel –, nem a fordítás, hanem a futtatás során kell beállítanunk. A dinamikus tömb bármikor átméretezhető, illetve törölhető.

### 10.4.1 Egydimenziós dinamikus tömbök

A dinamikus tömb deklarációja csak a tömb nevét és az elemeinek típusát tartalmazza:

```
var a: array of integer;
```

A deklaráció során csak egy hivatkozás (mutató) jön létre, magát a tömböt futás közben kell létrehoznunk a ***SetLength()*** eljárás hívásával:

```
SetLength(a,12);
```

A fenti utasítás hatására a memóriában létrejön a 12 egész típusú elemet tartalmazó tömb, melynek minden eleme 0 lesz. Az elemeket a szokásos indexeléssel érhetjük el, ahol az alsó indexhatár mindig 0 : *a[0], a[1], ..., a[11].*

A ***SetLenght()*** eljárás hívásával a tömböt bármikor átméretezhetjük. Az átméretezés során a tömb első elemei megmaradnak az alábbi példában látható módon:

```
var a,b: array of integer;
    n,i : integer;
begin
                        // A:                    B:
  n:=5;
  SetLength(a,n);       // A:0,0,0,0,0           B:
  for i:=0 to n-1 do
    a[i]:=sqr(i);       // A:0,1,4,9,16          B:
  SetLength(a,8);       // A:0,1,4,9,16,0,0,0    B:
  SetLength(a,4);       // A:0,1,4,9             B:
  SetLength(a,5);       // A:0,1,4,9,0           B:
  a:=copy(a,1,3);       // A:1,4,9               B:
  b:=a;                 // A:1,4,9               B:1,4,9
  b[1]:=3;              // A:1,3,9               B:1,3,9
  SetLength(a,0);       // A:                    B:1,3,9
  b:=nil;               // A:                    B:
end.
```

A ***Copy()*** függvény a dinamikus tömb adott indexű elemétől kezdődően, a harmadik paraméterben megadott számú elemet átmásol egy új dinamikus tömbbe:

```
dinamikus_tomb2 := Copy(dinamikus_tomb1,indextől,elemszám);
```

Azonos típusú dinamikus tömbök között értelmezett az értékadás művelete is, melynek eredményeként a bal oldalon álló dinamikus tömb mutatója ugyanarra a helyre hivatkozik, ahova a jobb oldalon álló dinamikus tömbbé. Ennek következtében például a *b[1]* elem megváltoztatása maga után vonja az *a[1]* megváltozását is, hiszen ugyanarra az elemre hivatkoznak. Amennyiben szeretnénk elkerülni a fentiekben vázolt működést, a példában használt *b:=a;* értékadás helyett az alábbi megoldások közül választhatunk:

```
b:=copy(a,0,length(a));
```

vagy

```
b:=a;
SetLength(a,length(a));
```

A dinamikus tömböt többféleképpen is törölhetjük a memóriából:

```
a:=nil;

SetLenght(a,0);

Finalize(a);
```

Dinamikus tömbök esetén a ***Length()*** függvény a tömb elemeinek számát adja meg, a ***Low()*** függvény az alsó indexhatárt adja vissza (ami mindig 0), míg a ***High()***függvény a felhasználható legnagyobb index értékével tér vissza. Nulla hosszú tömb esetén a ***Length()*** 0 értéket ad, míg a ***High()*** értéke -1 lesz.

### 10.4.2 Többdimenziós dinamikus tömbök

A többdimenziós dinamikus tömböket, mint dinamikus tömbök dinamikus tömbjének (stb.) dinamikus tömbjét állíthatjuk elő. Példaként tekintsük az alábbi kétdimenziós tömböt:

```
var
  c : array of array of integer;
```

A ***SetLength()*** hívásban, mind a két kiterjedésben meg kell adni a tömb méretét. Az alábbi utasítással egy 7x12-es méretű mátrix tárolására alkalmas tömböt hozunk létre:

```
SetLength(c,7,12);
```

A tömb elemeinek eléréséhez a sorokat *0..6*, míg az oszlopokat *0..11* tartományban indexszelhetjük.

A többdimenziós dinamikus tömbök méretét több lépésben is megadhatjuk, mint ahogy ez az alábbi példában is látható. A következőkben egy 5x5–ös méretű, bal alsó háromszögmátrix tárolására alkalmas adatstruktúrát hozunk létre:

```
var hm : array of array of integer;
    i, j: integer;
begin
  // Létrehozás
  SetLength(hm,5);              // a sorok száma
  for i:=0 to 4 do
     SetLength(hm[i], i+1);   // oszlopok száma soronként
  // Feltöltés
  for i:=0 to 4 do
   for j:=0 to i do
     hm[i,j]:=hm[i][j]+random(23);
  // Kiíratás
  for i:=0 to 4 do
   begin
     for j:=0 to i do
        write(hm[i,j]:5);
     writeln;
   end;
```

```
   12
   13   16
   19   13   19
   12   19    9   14
   14    8   10    6   20
```

```
  // Törlés
  Finalize(hm);
end.
```

A többdimenziós tömbök törlésére szintén a ***Finalize()*** eljárást, illetve a **nil** értékkel való értékadást használhatjuk.

## 10.5 Önhivatkozó adatstruktúrák

A dinamikus adatstruktúrák programozása során gyakori eset, hogy egy rekordban olyan mutatómezőt kell elhelyeznünk, amelyik magára a rekordra hivatkozik. Ebben az esetben a rekord definíciója előtt, ugyanabban a **type** szekcióban kell szerepeltetnünk a rekordra mutató pointer típusának deklarációját:

```
type
    pelem = ^elem;
    elem = record
        adat : integer;
        kovetkezo: pelem;
    end;
```

Az alábbiakban a lineáris lista példáján mutatjuk be az önhivatkozó adatstruktúrák alkalmazásának lépéseit.

```
program linearis_lista;

type
    pelem = ^elem;
    elem = record
        adat : integer;
        kovetkezo: pelem;
     end;

var
    kezdoptr : pelem = nil;
    segedptr : pelem = nil;
    i : integer;

begin
    // 12-elemű lista létrehozása és feltöltése
    randomize;
    for i:=1 to 12 do
      begin
        new(segedptr);    // helyfoglalás a rekord számára
        segedptr^.adat := random(123);
        segedptr^.kovetkezo := nil;
        if kezdoptr<>nil then
            segedptr^.kovetkezo:=kezdoptr;
        kezdoptr := segedptr;  // belépési pont
      end;
```

```
        // a lista adatainak megjelenítése
        segedptr:=kezdoptr;
        while segedptr <> nil do
          begin
            write(segedptr^.adat:4);
            segedptr := segedptr^.kovetkezo;
          end;
        writeln;

        // a lista elemeinek törlése
        while kezdoptr<>nil do
          begin
            segedptr := kezdoptr^.kovetkezo;
            dispose(kezdoptr);
            kezdoptr := segedptr;
          end;
        readln;
    end.
```

## 10.6 Feladatok a mutatók használatára

A mutatók használata minden programozási nyelv legnehezebben elsajátítható témaköre közé tartozik. Általában többszöri nekirugaszkodást igényel a biztonságos alkalmazás megtanulása. Bizonyos esetekben, például a dinamikus tömbök és a **class** osztályok bevezetésével, a Free Pascal elfedi előlünk a mutatókat, így sokkal kényelmesebb megoldásokat is használhatunk a programunkban. Sajnos ez nem minden esetben van így, mint ahogy azt az alábbi példaprogramok is mutatják.

A példaprogramok célja, a mutatók és a dinamikus tömbök kezelésével kapcsolatos kérdések tisztázása.

Írjunk programot, amely meghatározza két egész szám négyzetösszegének gyökét! (*ptrsqrt*)

```
program ptrsqrt;
var
   pa, pb: ^integer;
   pered: ^real;
begin
   new(pa); new(pb);
   new(pered);
   write('1.adat: '); readln(pa^);
   write('2.adat: '); readln(pb^);
   pered^ := sqrt(sqr(pa^)+sqr(pb^));
   writeln('eredmény: ',pered^:6:2);
   dispose(pa); dispose(pb);
   dispose(pered);
   readln;
end.
```

*A program futásának eredménye:*

```
1.adat: 3
2.adat: 4
eredmény:    5.00
```

Készítsünk programot, amely egy 20-elemű, egész típusú, dinamikus helyfoglalású tömböt feltölt adott darabszámú adattal, majd pedig kiszámítja a tömb elemeinek átlagát! Használjuk a ***new/dispose*** eljárásokat! (*newatlag*)

```
program newatlag;
type tomb = array[1..20] of integer;
var
   px : ^tomb;
   db, i: integer;
   atlag: real;
begin
   new(px); // helyfoglalás
   write('Adatok száma: ');
   readln(db);
   for i:=1 to db do
   begin
     write(i:2,'. adat: '); readln(px^[i]);
   end;
   atlag := 0;
   for i:=1 to db do
      atlag := atlag + px^[i];
   atlag := atlag/db;
   writeln('Az adatok átlaga: ',atlag:6:2);
   dispose(px); // a memória felszabadítása
   readln;
end.
```

*A program futásának eredménye:*

```
Adatok száma: 4
 1. adat: 2
 2. adat: 6
 3. adat: 4
 4. adat: 7
Az adatok átlaga:   4.75
```

Készítsünk programot, amely adott darabszámú, egész elemű, dinamikus helyfoglalású tömböt feltölt adatokkal, majd pedig kiszámítja a tömb elemeinek átlagát! Használjuk a ***getmem/freemem*** eljárásokat! (*getmematlag*)

```
{$R-}
program getmematlag;
type tomb = array[1..1] of integer; // csak a típusátalakításhoz kell
```

```
var
   px : pointer;
   db, i: integer;
   atlag: real;

begin
   write('Adatok száma: ');
   readln(db);
   getmem(px, db*sizeof(integer)); // helyfoglalás
   for i:=1 to db do
   begin
     write(i:2,'. adat: '); readln(tomb(px^)[i]);
   end;

   atlag := 0;
   for i:=1 to db do
      atlag := atlag + tomb(px^)[i];
   atlag := atlag/db;
   writeln('Az adatok átlaga: ',atlag:6:2);
   freemem(px); // a memória felszabadítása
   readln;
end.
```

*A program futásának eredménye:*

```
Adatok száma: 4
 1. adat: 2
 2. adat: 6
 3. adat: 4
 4. adat: 7
Az adatok átlaga:   4.75
```

Írjunk programot, amely adott darabszámú, egész elemű, dinamikus tömböt feltölt adatokkal, majd pedig kiszámítja a tömb elemeinek átlagát! (*dtombatlag*)

```
program dtombatlag;

var
   px : array of integer; // dinamikus tömb
   db, i: integer;
   atlag: real;

begin
   write('Adatok száma: ');
   readln(db);
   setlength(px, db); // helyfoglalás
   for i:=0 to high(px)do
   begin
     write(i:2,'. adat: '); readln(px[i]);
   end;

   atlag := 0;
   for i:=0 to high(px) do
      atlag := atlag + px[i];
   atlag := atlag/db;
   writeln('Az adatok átlaga: ',atlag:6:2);
```

```
   setlength(px,0); // a memória felszabadítása
   readln;
end.
```

*A program futásának eredménye:*

```
Adatok száma: 4
 1. adat: 2
 2. adat: 6
 3. adat: 4
 4. adat: 7
Az adatok átlaga:   4.75
```

Készítsünk programot, amely a komplex számokhoz definiálja a négy alapműveletet! A feladatot mutató-paramétereket használó alprogramokkal oldjuk meg! (*komplex4m*)

```
program komplex4m;
type
      PKomplex = ^Komplex;
      Komplex = record
         re,im: real;
      end;

  procedure Osszead(x,y:PKomplex; var ered:PKomplex);
  begin
     ered^.re :=  x^.re + y^.re;
     ered^.im :=  x^.im + y^.im;
  end;

  procedure Kivon(x,y:PKomplex; var ered:PKomplex);
  begin
     ered^.re :=  x^.re - y^.re;
     ered^.im :=  x^.im - y^.im;
  end;

  procedure Szoroz(x,y:PKomplex; var ered:PKomplex);
  begin
     ered^.re := x^.re * y^.re - x^.im * y^.im;
     ered^.im := x^.re * y^.im + x^.im * y^.re;
  end;

  procedure Oszt(x,y:PKomplex; var ered:PKomplex);
  begin
     ered^.re:=((x^.re*y^.re)+(x^.im*y^.im)) /
               ((y^.re*y^.re)+(y^.im*y^.im));
     ered^.im:=((x^.im*y^.re)+(-x^.re*y^.im)) /
               ((y^.re*y^.re)+(y^.im*y^.im));
  end;

  procedure Kiir(x,y:PKomplex;m: char;ered:PKomplex);
  begin
    write(x^.re:3:1,'+',x^.im:3:1,'i');
    write(' ',m,' ');
    write(y^.re:3:1,'+',y^.im:3:1,'i = ');
    writeln(ered^.re:5:1,'+',ered^.im:5:1,'i');
  end;
```

```
var
   Px1,Px2,Pered0: PKomplex;
begin
  new(Px1); new(Px2); new(Pered0);
  writeln;
  write('1. komplex szám valós része   : ');readln(Px1^.re);
  write('                képzetes része: ');readln(Px1^.im);
  write('2. komplex szám valós része   : ');readln(Px2^.re);
  write('                képzetes része: ');readln(Px2^.im);
  writeln;
  writeln('Műveletek');
  writeln('Az összeadás eredménye: ');
  Osszead(Px1,Px2,Pered0);
  Kiir(Px1,Px2,'+',Pered0);
  writeln('A kivonás eredménye: ');
  Kivon(Px1,Px2,Pered0);
  Kiir(Px1,Px2,'-',Pered0);
  writeln('A szorzás eredménye: ');
  Szoroz(Px1,Px2,Pered0);
  Kiir(Px1,Px2,'*',Pered0);
  writeln('A osztás eredménye : ');
  Oszt(Px1,Px2,Pered0);
  Kiir(Px1,Px2,'/',Pered0);
  dispose(Px1); dispose(Px2); dispose(Pered0);
  readln;
end.
```

*A program futásának eredménye:*

```
1. komplex szám valós része   : 1
                képzetes része: 1
2. komplex szám valós része   : 1
                képzetes része: 1

Műveletek
Az összeadás eredménye:
1.0+1.0i + 1.0+1.0i =   2.0+  2.0i
A kivonás eredménye:
1.0+1.0i - 1.0+1.0i =   0.0+  0.0i
A szorzás eredménye:
1.0+1.0i * 1.0+1.0i =   0.0+  2.0i
A osztás eredménye :
1.0+1.0i / 1.0+1.0i =   1.0+  0.0i
```

Írjunk programot, amely dinamikus rekordtömbben tárolja a diákok nevét és születési évét! Rendezzük az adatokat névsorba, illetve a diákok kora szerint növekvő sorrendbe! (*diakdtomb*)

```
program diakdtomb;
type
   tanulo = record
      nev    : string[32];
      szulev : integer;
   end;
```

```
   dtanulok = array of tanulo;
procedure olvas(var x : dtanulok);
var
   i, m : integer;
begin
   write('A tanulók száma: '); readln(m);
   setlength(x,m);
   for i:=0 to high(x) do
   begin
      writeln;
      write('Neve          : '); readln(x[i].nev);
      write('Születési éve: '); readln(x[i].szulev);
   end;
end;

procedure nevsorbarendez(var x : dtanulok);
var
   i,j: integer;
   s  : tanulo;
begin
   for i:=0 to high(x)-1 do
     for j:=i+1 to high(x) do
        if x[i].nev > x[j].nev then
           begin
              s := x[i];
              x[i]:= x[j];
              x[j]:=s;
           end;
end;

procedure szulevrendez(var x : dtanulok);
var
   i,j: integer;
   s  : tanulo;
begin
   for i:=0 to high(x)-1 do
     for j:=i+1 to high(x) do
        if x[i].szulev < x[j].szulev then
           begin
              s := x[i];
              x[i]:= x[j];
              x[j]:=s;
           end;
end;

procedure kiir(const x : dtanulok);
var
   i,n : integer;
begin
   for i:=0 to high(x) do
    begin
      n:= length(x[i].nev);
      writeln(x[i].nev,' ':32-n, x[i].szulev:8);
    end;
end;
```

```
var
   evfolyam : dtanulok;
begin
   olvas(evfolyam);
   writeln; writeln('Alapadatok');
   kiir(evfolyam);
   nevsorbarendez(evfolyam);
   writeln;
   writeln('Név szerint rendezett adatok');
   kiir(evfolyam); writeln;
   szulevrendez(evfolyam);
   writeln('Születési év szerint rendezett adatok');
   kiir(evfolyam);
   setlength(evfolyam, 0);
   readln;
end.
```

*A program futásának eredménye:*

```
A tanulók száma: 4

Neve          : Nagy Adrienn
Születési éve: 1987

Neve          : Tóth Natália
Születési éve: 1979

Neve          : Kiss Anna
Születési éve: 1990

Neve          : Kertész Júlia
Születési éve: 1975

Alapadatok
Nagy Adrienn                          1987
Tóth Natália                          1979
Kiss Anna                             1990
Kertész Júlia                         1975

Név szerint rendezett adatok
Kertész Júlia                         1975
Kiss Anna                             1990
Nagy Adrienn                          1987
Tóth Natália                          1979

Születési év szerint rendezett adatok
Kiss Anna                             1990
Nagy Adrienn                          1987
Tóth Natália                          1979
Kertész Júlia                         1975
```

Készítsünk mátrix determinánsát meghatározó programot! A mátrixot tároljuk a sorai-ra mutató pointerek segítségével! (*getmemmatrix*)

```
program getmemmatrix;
const
     soremeles = ^m^j; // mint #13#10
```

```
type
     sorptrtomb  = array[1..256] of pointer;
     sor         = array[1..256] of real;
     matrixptr   = ^sorptrtomb;

procedure kiir(n:byte; mat: matrixptr);
var
  i, j : integer;
begin
  writeln;
  for i:=1 to n do
    begin
      for j:=1 to n do
         write(sor(mat^[i]^)[j]:6:2);
      writeln;
    end;
  writeln;
end;

// Determináns-számító függvény
// Minden (al)mátrixban az első oszlop elemein haladunk végig -
// az almátrixot a 2. elemre mutató sorpointerekből állítjuk össze.
// Az eredeti mátrix elemeit nem másoljuk!

function det(melyseg:byte; mat:matrixptr):real;
var
  almatrix : matrixptr;
  soridx, i, j : byte;
begin
  if melyseg=2 then   // 2x2-es determináns
     exit (sor(mat^[1]^)[1]*sor(mat^[2]^)[2] -
           sor(mat^[1]^)[2]*sor(mat^[2]^)[1]);
  // Az almatrix az almátrix soraira mutató pointereket tartalmazza
  getmem(almatrix, pred(melyseg) * sizeof(pointer));
  det := 0.0;
  for soridx:=1 to melyseg do
    begin
      i:=1;
      for j:=1 to melyseg do
         if j<>soridx then
            begin
               sorptrtomb(almatrix^)[i] := @sor(mat^[j]^)[2];
               inc (i);
            end;
         if odd(soridx) then
           det:=det+sor(mat^[soridx]^)[1]*det(pred(melyseg),almatrix)
         else
           det:=det-sor(mat^[soridx]^)[1]*det(pred(melyseg),almatrix);
    end;
  freemem (almatrix,pred(melyseg)*sizeof(pointer));
end;

// Főprogram
var matrix : matrixptr;
    i, j, meret : byte;
```

```
begin
  writeln ('Determináns-számítás');
  writeln;
  write('Kérem a mátrix méretét (<=256) :'); readln (meret);
  // helyfoglalás a sorokra mutató pointerek vektorának
  getmem (matrix, sizeof(pointer) * meret);
  // helyfoglalás a sorok számára
  for i:=1 to meret do
     getmem (matrix^[i], meret * sizeof(real));
  writeln (soremeles+'A mátrix elemei soronként:'+soremeles);
  for i:=1 to meret do
    for j:=1 to meret do
      begin
        write ('Matrix[',i,',',j,'] : ');
        readln (sor(matrix^[j]^)[i]);
      end;
  writeln;
  kiir(meret, matrix);
  writeln ('A megadott mátrix determinánsa: ',
           det(meret,matrix):8:2,'!');
  // a sorok felszabadítása
  for i:=1 to meret do
     freemem (matrix^[i], meret * sizeof(real));
  // a sorokra mutató pointerek vektorának felszabadítása
  freemem (matrix, sizeof(pointer) * meret);
  readln;
end.
```

*A program futásának eredménye:*

```
Determináns-számítás
Kérem a mátrix méretét (<=256) :4
A mátrix elemei soronként:
Matrix[1,1] : 2
Matrix[1,2] : 0
Matrix[1,3] : 4
Matrix[1,4] : 3
Matrix[2,1] : -1
Matrix[2,2] : 2
Matrix[2,3] : 6
Matrix[2,4] : 1
Matrix[3,1] : 10
Matrix[3,2] : 3
Matrix[3,3] : 4
Matrix[3,4] : -2
Matrix[4,1] : 2
Matrix[4,2] : 1
Matrix[4,3] : 4
Matrix[4,4] : 0
  2.00 -1.00 10.00  2.00
  0.00  2.00  3.00  1.00
  4.00  6.00  4.00  4.00
  3.00  1.00 -2.00  0.00
A megadott mátrix determinánsa:   144.00!
```

Írjunk kétirányú, ciklikus listát kezelő programot egész számok tárolására! Valósítsuk meg az alapvető listaműveleteket! (*ketiranyu_lista*)

```
program ketiranyu_lista;

type
    adattipus = integer;

    plistaelem =^ listaelem;
    listaelem = record
        adat : adattipus;
        elozo, kovetkezo : plistaelem;
    end;

    dlista = record
        elemszam : integer;
        belepes, aktptr : plistaelem;
    end;

// A lista inicializálása
procedure letrehoz(var dl : dlista);
begin
    dl.elemszam := 0;
    dl.belepes := nil;
    dl.aktptr := nil;
end;

// Elemet illesztünk a listába a belépési pontba
procedure hozzaad(var dl : dlista; elem : adattipus);
var
   ptr : plistaelem;
begin
   // helyfoglalás a listaelem számára
   new(ptr);
   ptr^.adat := elem;
   ptr^.elozo := nil;
   ptr^.kovetkezo := nil;
   inc(dl.elemszam);
   if dl.belepes=nil then  // ha első elem
      begin
         dl.belepes := ptr;
         ptr^.elozo := ptr;
         ptr^.kovetkezo := ptr;
      end
   else  // nem első elem, befűzés a láncba
      begin
            ptr^.kovetkezo := dl.belepes;
            dl.belepes^.elozo^.kovetkezo := ptr;
            ptr^.elozo := dl.belepes^.elozo;
            dl.belepes^.elozo := ptr;
      end;
end;
```

```
// A lista elemeinek törlése
procedure torol(var dl : dlista);
var
   ptr1, ptr2 : plistaelem;
begin
   // szétvágjuk a listát
   ptr1 := dl.belepes;
   dl.belepes^.elozo^.kovetkezo := nil;
   // előre haladva töröljük az elemeket
   while ptr1^.kovetkezo <> nil do
      begin
         ptr2 := ptr1^.kovetkezo;
         dispose(ptr1);
         ptr1 := ptr2;
      end;
  dl.belepes := nil;
end;

// Adat keresése a listában, megadva az adatot tartalmazó
// listelem címét, vagy nil értéket, ha nincs meg az adat
function keres(const dl : dlista; elem : adattipus) : plistaelem;
var
   ptr1, ptr2 : plistaelem;
begin
   ptr1 := dl.belepes;
   ptr2 := nil;
   // üres lsitában nem keresünk
   if ptr1 = nil then exit(nil);
   // előre haladva megyünk végig a listán
   repeat
     if ptr1^.adat = elem then  // ha megtaláltuk az elemet
        begin
           ptr2 := ptr1;
           break;
        end
    else
        ptr1 := ptr1^.kovetkezo;
   until ptr1 = dl.belepes;
   keres := ptr2;
end;

// Az adott adatot tároló első elemet kivesszük a listából
procedure kivesz(var dl :dlista; elem : adattipus);
var
   ptr : plistaelem;
begin
   // megkeressük az elemet
   ptr := keres(dl, elem);
   // ha nincs benne, semmi dolgunk
   if ptr = nil then exit;
   // ha a belépési pontban áll az elem, a következő elem
   // lesz a belépési pont
   if ptr = dl.belepes then
       dl.belepes := dl.belepes^.kovetkezo;
```

```
    if ptr^.kovetkezo = ptr then  // ha az utolsó elem, töröljük azt
        dl.belepes := nil
    else
        begin                          // ha nem, kifűzzük az elemet
           ptr^.elozo^.kovetkezo := ptr^.kovetkezo;
           ptr^.kovetkezo^.elozo := ptr^.elozo;
        end;
    // az iteráció mutatójának érvénytelenítése
    dl.aktptr := nil;
    // töröljük az elemet
    dispose(ptr);
    dec(dl.elemszam);
end;

// Az elemek megjelenítése
procedure kiir(const dl : dlista; irany : boolean);
var
    ptr1 : plistaelem;
begin
    write('<< ');
    ptr1 := dl.belepes;
    // üres lista
    if ptr1 <> nil then // előre haladva megyünk végig a listán
       repeat
          write(ptr1^.adat:5);
          if irany then
              ptr1 := ptr1^.kovetkezo
          else
              ptr1 := ptr1^.elozo;
       until ptr1 = dl.belepes ;
    writeln('>>');
end;

// A listát előre (true) /vissza (false) haladva bejáró iterátor
function kovelozo(var dl : dlista; irany : boolean) : adattipus;
begin
    if dl.aktptr = nil then
      dl.aktptr := dl.belepes
    else
      if irany then
          dl.aktptr := dl.aktptr^.kovetkezo
       else
          dl.aktptr := dl.aktptr^.elozo;
    // kivédjük az üres lista esetét
    if dl.aktptr<>nil then
        kovelozo := dl.aktptr^.adat
    else
        kovelozo := 0;
end;

var
  lista : dlista;
  i, db : integer;
  elem  : adattipus;
```

```
begin
  randomize;
  letrehoz(lista);
   // feltöltjük a listát véletlen számokkal
  for i:= 1 to 12 do
    hozzaad(lista, random(123));
  writeln;
  writeln('A lista ', lista.elemszam,' elemet tartalmaz:');
  kiir(lista, true);
  writeln;
  writeln('A lista elemei forditott sorrendben:');
  kiir(lista, false);

  // kiszedjük a listából a páros elemeket
  // legfeljebb ennyi vizsgálatot kell végeznünk, mivel az
  // iterátor minden törlés után újraindul
  db := sqr(lista.elemszam);
  for  i:=1 to db do
     begin
        elem := kovelozo(lista,true);
        if elem mod 2 = 0 then kivesz(lista, elem);
     end;
  writeln;
  writeln('A páratlan elemek listája ', lista.elemszam,
          ' elemet tartalmaz:');
  kiir(lista, true);

  torol(lista);
  kiir(lista, true);
  readln;
end.
```

*A program futásának eredménye:*

```
A lista 12 elemet tartalmaz:
<<   112   20  119   23   76   43   47   87   33  116   28    8>>
A lista elemei forditott sorrendben:
<<   112    8   28  116   33   87   47   43   76   23  119   20>>
A páratlan elemek listája 6 elemet tartalmaz:
<<   119   23   43   47   87   33>>
<< >>
```

Készítsünk programot, amely a megadott egész számokat tökéletesen kiegyensúlyozott bináris fa csomópontjaiban tárolja, és bejárva a fát összegzi a számokat! (*faepito*)

```
program faepito;

type ref = ^csomopont;
     csomopont = record
        kulcs:integer;
        bal, jobb : ref;
     end;
```

```
// Az n csomópontú, tökéletesen kiegyensúlyozott fa építése
function fa(n : integer): ref;
var ujcsomopont : ref ;
    x,nb,nj : integer;
begin
   if n=0 then fa:=nil
   else
      begin
        nb := n div 2;
        nj := n - nb - 1;
        read(x);
        new(ujcsomopont);
        with ujcsomopont^ do
           begin
              kulcs := x ;
              bal   := fa(nb);  jobb  := fa(nj)
           end;
           fa:=ujcsomopont
      end
end;

// A fa megjelenítése, h távolsággal az egyes szintek között
procedure fanyomtato( t : ref; h:integer);
var i:integer;
begin
   if t<>nil then
      with t^ do
         begin
           fanyomtato(bal, h+1);
           for i:=1 to h do write(#32 : 8);
           writeln(kulcs:8);
           fanyomtato(jobb, h+1);
         end
end;

// A csomópontok értékének összegzése
procedure faosszeg(t : ref; var sum:longint);
begin
    if t<>nil then
      begin
        sum:=sum + t^.kulcs;
        faosszeg(t^.bal,sum);
        faosszeg(t^.jobb,sum);
      end;
end;

// A fa lebontása
procedure fatorles(t : ref);
begin
    if t<>nil then
      begin
        fatorles(t^.bal);   fatorles(t^.jobb);
        dispose(t);
      end;
end;
```

```
var
     gyoker : ref;
     s : longint = 0;
     n : integer;
begin
   write('A csomópontok száma: '); readln(n);
   writeln('A csomópontok adatai: ');
   gyoker := fa(n);
   fanyomtato(gyoker,2);
   faosszeg(gyoker, s);
   writeln('A csomópontértékek összege :', s:8);
   fatorles(gyoker);
   readln;
   readln;
end.
```

*A program futásának eredménye:*

```
A csomópontok száma: 9
A csomópontok adatai:
12 23 7 11 30 10 2 79 19
                                        11
                                    7
                            23
                                    30
                    12
                                        79
                                    2
                            10
                                    19
A csomópontértékek összege :      193
```

Írjunk rekurzív bináris fát, amely növekvő sorrendbe rendezi a beolvasott egész számokat, és ellenőrzi, hogy az adat eleme-e a fának! (*binfa*)

```
program binfa;
type
  TElem = Word;
  PCsomo = ^TCsomo;
  TCsomo = record
    Bal: PCsomo;
    Elem: TElem;
    Jobb:PCsomo;
  end;

procedure KeresBeszur (Elem: TElem; var Gyoker: PCsomo);
begin
  if Gyoker = nil then
   begin
     New(Gyoker);
     Gyoker^.Elem := Elem;
     Gyoker^.Bal  := nil;
     Gyoker^.Jobb := nil;
   end
```

```
  else
   begin
     if Elem < Gyoker^.Elem then
        KeresBeszur(Elem,Gyoker^.Bal)
     else
        KeresBeszur(Elem,Gyoker^.Jobb)
   end;
end;

procedure Rendez(Gyoker:PCsomo);
begin
  if Gyoker <> nil then
  begin
    Rendez(Gyoker^.Bal);
    write(Gyoker^.Elem:4);
    Rendez(Gyoker^.Jobb);
  end;
end;

function Keres(Elem: TElem; Gyoker:PCsomo):PCsomo;
begin
  if Gyoker = nil then
     Keres := nil
  else If Elem < Gyoker^.Elem then
    Keres:= Keres(Elem,Gyoker^.Bal)
  else if Elem > Gyoker^.Elem then
    Keres:= Keres(Elem,Gyoker^.Jobb)
  else
    Keres:= Gyoker;
end;

var
   Fa:PCsomo;
   Elem: TElem;
   Csomo:Pcsomo;
begin
  Fa := nil;
  write('Elem (kilépés: 0): '); Readln(Elem);
  while Elem <> 0 do
  begin
    KeresBeszur(Elem,Fa);
    write('Elem: ');
    Readln(Elem);
  end;
  writeln('Rendezett lista: '); Rendez(Fa); Writeln;
  write('Elem: '); readln(Elem);
  Csomo:= Keres(Elem,Fa);
  if Csomo = nil then
    writeln('Nincs ilyen elem')
  else
    writeln('Létezik az elem: ',Csomo^.Elem);
  readln;
end.
```

*A program futásának eredménye:*

```
Elem (kilépés: 0): 12
Elem: 3
Elem: 45
Elem: 2
Elem: 67
Elem: 14
Elem: 32
Elem: 1
Elem: 88
Elem: 0
Rendezett lista:
   1   2   3  12  14  32  45  67  88
Elem: 12
Létezik az elem: 12
```

# 11. Objektum-orientált programozás

Az objektum-orientált programozás (*Object-Oriented Programming, OOP*) az objektum bevezetésével megszünteti a program kódjának és adatainak szétválasztását. Az objektumok használata lehetővé teszi az egyszerű, jól kézben tartható programok készítését. Az objektum-orientált nyelvek három főbb jellegzetessége:

- az egységbezárás (*encapsulation*),
- az öröklés (*inheritance*),
- a sokalakúság (*polymorphism*).

## 11.1 Az egységbezárás

Az objektum-orientált nyelveknek egyik legfontosabb tulajdonsága az adatok és az adatokon végzett műveletek egybeépítése (*encapsulation*). Az adatok és a műveletek egyenrangúak, és zárt programegységet alkotnak. A zárt egységet alkotó adattípust Free Pascalban objektumnak (**object**), az objektum adatait (mezőit) kezelő alprogramokat pedig metódusoknak nevezzük. A OOP-ben elfogadott szóhasználat szerint az **object** típussal az objektum osztályát (típusát) hozzuk létre, míg az objektumok az osztály típusú változók.

### 11.1.1 Az object típus

Az objektumtípus készítése sokban emlékeztet a **record** típus kialakítására. Alapvető különbség, hogy ebben az esetben az **object** foglalt szót kell használnunk, és a deklarációban eljárások és függvények (metódusok) fejlécei is megjelenhetnek. Az objektumtípus teljes megadásához azonban a metódusokat is definiálnunk kell.

```
type
   objektumtípus = ⟨packed⟩ object
      mező1: típus1;
    ...
      mezőN: típusN;
      metódus1 fejléce;
    ...
      metódusM fejléce;
   end;

Metódusdefiníciók
```

A metódusdefiníciókban az objektum nevét ponttal elválasztva kapcsoljuk a metódus nevéhez. Erre azért van szükség, hogy a különböző objektumok, azonos nevű metódusait egyértelműen megkülönböztessük egymástól. Felhívjuk a figyelmet arra, hogy a metódusparaméterek neve nem egyezhetnek meg az adatmezők nevével.

Tervezzünk objektumtípust, az összeadás, kivonás és szorzás alapműveleteket megvalósító kalkulátor programozásához! (*kalkulatortipus*)

Az adatmezőkben a bemeneti adatok mellett az eredményt is eltároljuk:

| | |
|---|---|
| *a, b* | a valós operandusok, |
| *muv_jel* | a karakter típusú műveleti jel lehetséges értékei: +,-,*, |
| *eredmeny* | a művelet eredményét tároló valós mező. |

A metódusok feladata az objektumtípusú változók inicializálása, valamint a kijelölt művelet elvégzését követően, az eredmény visszaadása.

| | |
|---|---|
| `procedure Init(a0,b0:real; muv_jel0:char);` | a metódus értéket ad az operandusoknak és a műveleti jelnek, |
| `function Muvelet:real;` | a *Muvelet()* metódus elvégzi az operátor által kijelölt műveletet. |

A *Kalkulator* típus:

```
type
    Kalkulator = object
       // adatmezők
       a,b      : real;
       muv_jel  : char;
       eredmeny: real;
       // metódusok
       procedure Init(a0,b0:real; muv_jel0:char);
       function Muvelet:real;
    end;
```

Az objektum (teljes) definiálásához, a metódusokat is meg kell adnunk.

```
procedure Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   a        := a0;
   b        := b0;
   muv_jel  := muv_jel0;
   eredmeny:= 0;
end;

function Kalkulator.Muvelet:real;
begin
   case muv_jel of
     '+' : eredmeny:= a+b;
     '-' : eredmeny:= a-b;
     '*' : eredmeny:= a*b;
   end;
   Muvelet:= eredmeny;
end;
```

### 11.1.2 Az objektumpéldányok létrehozása

Az objektumpéldányokat (az objektum típusú változókat) – a közönséges változókhoz hasonlóan – statikus vagy dinamikus helyfoglalással egyaránt létrehozhatjuk. Bármelyik módszert is választjuk, az objektumpéldányok mindegyike saját adatmezőkkel (adatterülettel) rendelkezik, azonban a metódusokat megosztva (közösen) használják.

A kérdés már csak az, honnan tudja egy metódus, hogy éppen melyik objektumpéldányon kell elvégeznie a műveletét? A megoldás a fordító „kezében" van, hisz minden metódust kiegészít egy nem látható, objektumtípusú változóparaméterrel, melynek neve **self** (önmaga), és amelyben az aktuális objektumpéldányt adja át a metódusnak. A metódusokon belül az adatmezők neveit pedig a **self** paraméterrel minősítve fordítja. A **self** paramétert ugyan nem látjuk, azonban szükség esetén magunk is felhasználhatjuk:

```
procedure Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   self.a   := a0;
   self.b   := b0;
   self.muv_jel := muv_jel0;
   self.eredmeny:= 0;
end;
```

Ha valamely metóduson belül az objektumpéldány címére van szükségünk, akkor a *@Self* kifejezést használhatjuk.

#### *11.1.2.1 Statikus helyfoglalású objektumpéldányok*

A statikus helyfoglalású objektumpéldányok a korábbi fejezetekben ismertetett élettartam szabályok szerint jönnek létre, illetve semmisülnek meg. A deklarálás helyétől függően a példányok lehetnek globálisak, vagy helyiek (lokálisak) valamely alprogramra nézve.

```
var
   k1, k2 : Kalkulator;
```

A rekordoknál megismert formában akár kezdőértékkel is elláthatjuk az egyes példányokat:

```
const
    k1: Kalkulator = (a:12; b:23; muv_jel:'+'; eredmeny:0);
var
    k2: Kalkulator = (a:19; b:79; muv_jel:'*'; eredmeny:0);
```

Az objektumtípus adatmezőit és metódusait szintén a rekordoknál bemutatott módon érhetjük el. Megjegyezzük, hogy az adatmezők közvetlen elérése ellentmond az OOP

alapelveinek, de csak a későbbiekben érvényesítjük ezeket az elveket. A hivatkozáshoz egyaránt használhatjuk a pontot, és a **with** utasítást (metódusok esetén is!):

```
begin
  k1.a := 12;
  k1.b := 23;
  k1.muv_jel:='-';
  writeln(k1.Muvelet);
  k2.Init(2, 3, '*');
  writeln(k2.Muvelet);
end.
```

```
begin
  with k1 do
    begin
      a := 12;
      b := 23;
      muv_jel:='-';
      writeln(Muvelet);
      k2.Init(2, 3, '*');
      writeln(k2.Muvelet);
    end;
end.
```

Az azonos típusú objektumpéldányok az értékadó utasítás (:=) mindkét oldalán szerepelhetnek. Az értékadás során a jobb oldalon álló objektumváltozó teljes adatterülete a bal oldalon megadott objektumpéldány adatterületére másolódik:

```
k1.Init(12, 23, '*');
k2 := k1;
```

A fentiekben definiált *Kalkulator* típust egészítsük ki, a statikus objektumpéldányok használatát bemutató főprogrammal! (*statkalk*)

```
program statkalk;
type
   Kalkulator = object
      a,b      : real;
      muv_jel  : char;
      eredmeny: real;
      procedure Init(a0,b0:real; muv_jel0:char);
      function Muvelet:real;
   end;

procedure Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   a        := a0;
   b        := b0;
   muv_jel  := muv_jel0;
   eredmeny:= 0;
end;

function Kalkulator.Muvelet:real;
begin
   case muv_jel of
     '+' : eredmeny:= a+b;
     '-' : eredmeny:= a-b;
     '*' : eredmeny:= a*b;
   end;
   Muvelet:= eredmeny;
end;
```

```
var
   x,y: real;
   m: char;
   k : Kalkulator;

begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+','-','*'];

   k.Init(x,y,m);
   with k do
     writeln(a:5:2, muv_jel:2, b:5:2,' =', Muvelet:6:2);

   k.Init(5,6,'*');
   writeln(k.a:5:2, k.muv_jel:2, k.b:5:2,' =', k. Muvelet:6:2);
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 12
2. adat: 23
műveleti jel (+,-,*): +
12.00 +23.00 = 35.00
 5.00 * 6.00 = 30.00
```

### *11.1.2.2 Dinamikus helyfoglalású objektumpéldányok*

A dinamikus objektumpéldányok létrehozásához mutatókat használunk:

```
type
     pKalkulator = ^Kalkulator;

var
     pk : ^Kalkulator;
     pk1 : pKalkulator;
```

Az objektumpéldányokat a ***new()*** alprogram hívásával hozzuk létre, melyet eljárásként és függvényként egyaránt hívhatjuk:

```
new(pk);

pk1 := new(PKalkulator);
```

Első esetben a lefoglalt tárterület címe közvetlenül a *pk* változóba töltődik, míg a második formában a ***new()*** függvényértékként adja vissza a memóriacímet.

A mutató által kijelölt dinamikus objektumpéldányokra szintén kétféleképpen hivatkozhatunk, a pont jellel, illetve a **with** utasítás segítségével. Mindkét esetben a mutató nevét követően alkalmaznunk kell az előző fejezetben megismert ^ jelet:

```
begin
  pk^.a := 12;
  pk^.b := 23;
  pk^.muv_jel:='-';
  writeln(pk^.Muvelet);
end.
```

```
begin
  with pk^ do
    begin
      a := 12;
      b := 23;
      muv_jel := '-';
      writeln(Muvelet);
    end;
end.
```

Ha nincs már szükségünk az objektumpéldányra, akkor a ***dispose()*** eljárás segítségével felszabadítjuk a lefoglalt memóriaterületet:

```
dispose(pk);
```

A fentiekben definiált *Kalkulator* típust egészítsük ki a dinamikus objektumpéldányok használatát bemutató főprogrammal! (*dinkalk*)

```
program dinkalk;
type
   Kalkulator = object
      a,b      : real;
      muv_jel  : char;
      eredmeny: real;
      procedure Init(a0,b0:real; muv_jel0:char);
      function Muvelet:real;
   end;

procedure Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   a        := a0;
   b        := b0;
   muv_jel  := muv_jel0;
   eredmeny:= 0;
end;

function Kalkulator.Muvelet:real;
begin
   case muv_jel of
     '+' : eredmeny:= a+b;
     '-' : eredmeny:= a-b;
     '*' : eredmeny:= a*b;
   end;
   Muvelet:= eredmeny;
end;

type
     pKalkulator = ^Kalkulator;

var
     pk  : ^Kalkulator;
     pk1 : pKalkulator;
     x,y : real;
     m   : char;
```

```
begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+','-','*'];
   new(pk);     // memóriafoglalás
   pk^.Init(x,y,m);
   with pk^ do
     writeln(a:5:2, muv_jel:2, b:5:2,' =', Muvelet:6:2);
   pk^.Init(5,6,'*');
   writeln(pk^.a:5:2, pk^.muv_jel:2, pk^.b:5:2,' =', pk^.Muvelet:6:2);
   dispose(pk);  // a lefoglalt tarterület felszabadítása
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 12
2. adat: 23
műveleti jel (+,-,*): +
12.00 +23.00 = 35.00
 5.00 * 6.00 = 30.00
```

### 11.1.3 A konstruktor és a destruktor

Az eddigi példáinkból láthatjuk, hogy az objektumpéldányokat a létrehozásuk után inicializálni kell, függetlenül a létrehozás módjától (statikus, dinamikus). Ezt a feladatot normál (eljárás) metódusokkal, és különleges ún. konstruktor (**constructor**) metódusokkal egyaránt megoldhatjuk.

Amikor egy objektumpéldány inicializálása során erőforrásokat foglalunk (memóriát, állományt stb.), akkor ezeket a példány megszűnése előtt el kell engednünk (fel kell szabadítani, le kell zárni stb.). Ezeket a műveleteket, a normál metódusok mellett, az ún. destruktor (**destructor**) metódusokban is elvégezhetjük.

A konstruktorok gyakran az *Init*, míg a destruktorok a *Done* nevet viselik. Természetesen mindkét metódusfajta esetén alkalmazhatjuk az túlterhelés (*overloading*) mechanizmusát.

A dinamikus helyfoglalású objektumpéldányok konstruktorát és destruktorát a memóriakezelő ***new()*** és ***dispose()*** alprogramok is aktivizálhatják, amihez az eljárások bővített formáját kell használnunk:

```
new(objektummutató, konstruktorhívás);
objektummutató := new(objektummutató_típusa, konstruktorhívás);
dispose(objektummutató, destruktorhívás);
```

Amennyiben a *PObjektum*, egy *Init()* konstruktorral rendelkező *TObjektum* osztály mutatótípusa, akkor az alábbi utasítások (soronként) teljesen azonos hatásúak.

```
var
    po : PObjektum;
begin
    po := new(PObjektum, Init)
    new(po, Init);
    new(po);  po^.Init;
end.
```

A ***dispose()*** esetében az alábbi két megoldás vezet azonos eredményre, ha a *Done()* destruktora az objektumtípusnak:

```
dispose(po, Done);
po^.Done; dispose(po);
```

Az elmondottak ellenére a konstruktorok és a destruktorok használata csak akkor kötelező, ha az objektumtípusunk az öröklés mechanizmusa során lecserélhető (ún. virtuális) metódusokat tartalmaz. A polimorfizmus megvalósításához szükséges metódustáblázatok felépítése és kitöltése a konstruktorok első hívásakor megy végbe. Az öröklléssel és a polimorfizmussal néhány alfejezettel később ismertetjük meg az Olvasót.

A *statkalk* példaprogram szereplő *Kalkulator* objektumtípus valós adatmezőit tegyük dinamikussá, és készítsünk a kezelésükhöz konstruktort és destruktort! (*konstdest*)

```
program konstdest;

type
   preal = ^real;
   Kalkulator = object
      a,b     : preal;
      muv_jel : char;
      eredmeny: preal;
      constructor Init(a0,b0:real; muv_jel0:char);
      destructor Done;
      function Muvelet:real;
      procedure Megjelenit;
   end;

constructor Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   new(a); new(b); new(eredmeny);
   a^ := a0;  b^ := b0;
   muv_jel  := muv_jel0;
   eredmeny^:= 0;
end;
```

```
destructor Kalkulator.Done;
begin
   dispose(a); dispose(b); dispose(eredmeny);
end;

function Kalkulator.Muvelet:real;
begin
   case muv_jel of
     '+' : eredmeny^:= a^+b^;
     '-' : eredmeny^:= a^-b^;
     '*' : eredmeny^:= a^*b^;
   end;
   Muvelet:= eredmeny^;
end;

procedure Kalkulator.Megjelenit;
begin
   writeln(a^:5:2, muv_jel:2, b^:5:2,' =', Muvelet:6:2);
end;

type
   PKalkulator = ^Kalkulator;
var
   x,y: real;
   m: char;
   k  : Kalkulator;
   pk : PKalkulator;

begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+', '-' ,'*'];

   k.Init(x,y,m);      // statikus példány konstruktora
   k.Megjelenit;
   k.Done;             // statikus példány destruktora

                       // dinamikus példány konstruktora
   pk := new(PKalkulator,Init(2*x,3*y, '-'));
   pk^.Megjelenit;
   dispose(pk,Done);   // dinamikus példány destruktora

   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 23
2. adat: 12
műveleti jel (+,-,*): +
23.00 +12.00 = 35.00
46.00 -36.00 = 10.00
```

### 11.1.4 Az adatrejtés elve

Az objektum-orientált programozás egyik alapgondolata az adatrejtés (*data hiding*), ami azt jelenti, hogy az objektum bizonyos részeihez (adatmezőihez, metódusaihoz) kívülről nem engedünk közvetlen, ellenőrizetlen hozzáférést.

Az elérhetőség alapján az objektumtípus adatmezőit és metódusait csoportokba gyűjthetjük, és megadhatjuk az egyes csoportokra vonatkozó hozzáférést. A **private** (saját) foglalt szóval tilthatjuk, míg az alapértelmezett **public** (nyilvános) foglalt szóval engedélyezhetjük a közvetlen hozzáférést. A **protected** (védett) foglalt szót öröklés esetén alkalmazzuk, melynek hatása a külvilág felé megegyezik a **private** szóéval.

Az objektumon belül a **private, public** és **protected** részek száma és sorrendje tetszőleges, azonban minden egyes részben először az adatmezőket kell megadnunk.

Az általános szabály, hogy az objektumok adatmezőit kívülről csak a metódusok hívásával érjük el, ezért az adatmezőket az objektumtípus **private**, a metódusokat pedig a **public** részében adjuk meg. Amennyiben egy adatmező értékét le kívánjuk kérdezni (*Get*) és/vagy módosítani (*Set*) szeretnénk, ún. elérési metódusokkal látjuk el az objektumtípust.

Az elmondottak figyelembevételével a *Kalkulator* típusunk bővített változata:

```
type
   Kalkulator = object
     private
       a,b      : real;
       muv_jel  : char;
       eredmeny: real;
     public
       constructor Init(a0,b0:real; muv_jel0:char);
       destructor Done;
       function Muvelet:real;
       procedure Megjelenit;
       function Get_adat1:real;
       function Get_adat2:real;
       function Get_muvjel:char;
       procedure Set_adat1(a0 : real);
       procedure Set_adat2(b0 : real);
       procedure Set_muvjel(muv_jel0 : char);
   end;
```

A Free Pascalban az elérési előírások önmagukban hatástalanok. Ahhoz, hogy az adatrejtés elve valóban érvényesüljön, az objektumtípust egy **unit** interfész részébe, míg a metódusok definícióját a modul implementációs részébe kell helyeznünk. Ezzel a megoldással az adatrejtés elvének megvalósulása mellett, a programunk felépítése is strukturáltabb lesz. Egyetlen modul több osztálydefiníciót is tárolhat, melyek között

ún. baráti (*friend*) viszony áll fenn, vagyis a metódusaik – szükség esetén – korlátozás nélkül elérhetik egymás példányinak adatmezőit.

A kibővített *Kalkulator* objektumtípust tároljuk *KalkUnit* modulban! Készítsük el az objektumpéldányok használatát bemutató programmodult is! (*\BovitettKalkulator*)

Az osztálydefiníciót tartalmazó *KalkUnit.pas* állomány csak **interface** és **implementation** részekből áll.

```
unit KalkUnit;

interface

type
   Kalkulator = object
     private
       a,b      : real;
       muv_jel  : char;
       eredmeny: real;
     public
       constructor Init(a0,b0:real; muv_jel0:char);
       destructor Done;
       function Muvelet:real;
       procedure Megjelenit;
       function Get_adat1:real;
       function Get_adat2:real;
       function Get_muvjel:char;
       procedure Set_adat1(a0 : real);
       procedure Set_adat2(b0 : real);
       procedure Set_muvjel(muv_jel0 : char);
   end;

implementation

constructor Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   a        := a0;
   b        := b0;
   muv_jel  := muv_jel0;
   eredmeny:= 0;
end;

function Kalkulator.Muvelet:real;
begin
   case muv_jel of
     '+' : eredmeny:= a+b;
     '-' : eredmeny:= a-b;
     '*' : eredmeny:= a*b;
   end;
   Muvelet:= eredmeny;
end;
```

```
procedure Kalkulator.Megjelenit;
begin
   writeln(a:5:2, muv_jel:2, b:5:2,' =', Muvelet:6:2);
end;

destructor Kalkulator.Done;
begin
   // Semmi teendőnk
end;

function Kalkulator.Get_adat1:real;
begin
   Get_adat1:=a;
end;

function Kalkulator.Get_adat2:real;
begin
   Get_adat2:= b;
end;

function Kalkulator.Get_muvjel:char;
begin
   Get_muvjel:= muv_jel;
end;

procedure Kalkulator.Set_adat1(a0 : real);
begin
   a := a0;
end;

procedure Kalkulator.Set_adat2(b0 : real);
begin
   b := b0;
end;

procedure Kalkulator.Set_muvjel(muv_jel0 : char);
begin
   muv_jel:= muv_jel0;
end;

end.
```

A *Kalkulator* objektumtípus használatát szemléltető főprogramot, a *KalProg.pas* fájlban találjuk:

```
program kalkprog;
uses kalkunit;

var
   x,y: real;
   m: char;
   k : Kalkulator;

begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
```

```
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+','-','*'];
   k.Init(x,y,m);
   with k do
      Megjelenit;
   k.Init(k.Get_adat1*2, k.Get_adat2*3,'*');
   k.Megjelenit;
   k.Set_adat1(2);
   k.Set_muvjel('-');
   k.Megjelenit;
   k.Done;
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 7
2. adat: 3
műveleti jel (+,-,*): *
 7.00 * 3.00 = 21.00
14.00 * 9.00 =126.00
 2.00 - 9.00 = -7.00
```

## 11.2 Az öröklés

Az objektum-orientált nyelvek másik fontos sajátossága az öröklés (*inheritance*). Ennek lényege, hogy egy meglévő objektumtípusból kiindulva, újabb objektumtípust készítünk, amely megkapja a felhasznált típus (nyilvános és védett) adatmezőit valamint metódusait. Azt az objektumtípust, amelytől az új objektumtípust származtatjuk, ős (*ancestor*) osztálynak, az új objektumtípust pedig származtatott (*descendant*) osztálynak nevezzük.

Az ősobjektum típusát a származtatott osztály definíciójában, az **object** foglalt szó után, zárójelben kell megadnunk:

```
type
   objektumtípus = ⟨packed⟩ object(őstípus)
    private
      meződefiníciók;
      metódusfejlécek;
    protected
      meződefiníciók;
      metódusfejlécek;
    public
      meződefiníciók;
      metódusfejlécek;
      ...
   end;

Metódusdefiníciók
```

Az öröklés során új adatmezőkkel és metódusokkal bővíthetjük a származtatott objektumtípust, azonban lehetőség van az örökölt metódusok azonos fejléccel, de új tartalommal történő újradefiniálására (*redefine*) is.

Free Pascalban egy objektumnak legfeljebb egy őse lehet, vagyis csak az egyszeres öröklést támogatja. Öröklés során az inicializáció fogalmát ki kell bővítenünk, mivel minden objektumnak saját magán kívül, gondoskodnia kell a közvetlen őse inicializálásáról is. A közvetlen ős metódusait kétféleképpen is hívhatjuk:

```
inherited metódushívás;
közvetlen_ős_objektumtípus.metódushívás;
```

Az elmondottakat jól szemlélteti az alábbi egyszerű, kétszintes objektum-hierarchia:

```
type   // Az ősosztály definíciója
     TOs = object
        private
          a : integer;
        public
          constructor Init(a0 : integer);
          procedure Kiir;
     end;

constructor TOs.Init(a0 : integer);
begin
  a := a0;
end;

procedure TOs.Kiir;
begin
  writeln('a= ', a);
end;

type   // A származtatott osztály definíciója
     TUtod = object(TOs)
        private
          b : integer;
        public
          constructor Init(a0 : integer; b0 : integer);
          procedure Kiir;
     end;

constructor TUtod.Init(a0 : integer; b0 : integer);
begin
  inherited Init(a0);  // TOs.Init(a0);
  b := b0;
end;

procedure TUtod.Kiir;
begin
  inherited;           // TOs.Kiir;
  writeln('b= ', b);
end;
```

```
var
  x : TOs;
  y : TUtod;
begin
  x.Init(7);
  x.Kiir;                  // a= 7
  y.Init(12,23);
  y.Kiir;                  // a= 12 / b= 23
  x := y;
  x.Kiir;                  // a= 12
  readln;
end.
```

Felhívjuk a figyelmet az *x:=y;* értékadásra. Általánosan is elmondhatjuk, hogy a származtatott objektum minden formában helyettesítheti tetszőleges ősét.

A példában a *TOs* objektumtípust kétféle szerepben jelent meg. Egyrészt változót (*x*) deklaráltunk vele, objektumpéldányt hoztunk létre – vagyis a *TOs* felhasználói voltunk. Másrészt pedig – az öröklést alkalmazva – új osztályt (*TUtod*) származtattunk belőle, vagyis továbbfejlesztői voltunk a *TOs* osztálynak. Ez a kettősség a **protected** hozzáférést is jellemzi. Egy objektumtípus felhasználói számára a **protected**, mint **private** jelenik meg, azonban a továbbfejlesztőknek **public** elérést biztosít.

A *Kalkulator* objektumtípusból származtassuk az *UjKalkulator* típust, amely a *Muvelet()* metódust újradefiniálva biztosítja az osztás elvégzését! (*ujkalkprg*)

A *Kalkulator* objektumtípus eredeti változatában a *Megjelenit()* metódusból hívjuk a *Muvelet()* metódust. Az objektumok ***zártsága*** miatt, az ősben definiált *Megjelenit()* eljárás csak az ős *Muvelet()* függvényét hajlandó meghívni. Mivel a megoldást a problémára csak a következő alfejezet tartalmazza, a példában eljárásként hívjuk a *Muvelet()* metódust, a *Megjelenit()* pedig az *eredmeny* adatmezőből veszi a művelet eredményét.

```
program ujkalkprg;
//  ----------  a Kalkulator osztály  ----------
type
   Kalkulator = object
     protected
       a,b      : real;
       muv_jel  : char;
       eredmeny: real;
     public
       constructor Init(a0,b0:real; muv_jel0:char);
       destructor Done;
       function Muvelet:real;
       procedure Megjelenit;
   end;
```

```
constructor Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   a       := a0;
   b       := b0;
   muv_jel := muv_jel0;
   eredmeny:= 0;
end;

function Kalkulator.Muvelet:real;
begin
   case muv_jel of
     '+' : eredmeny:= a+b;
     '-' : eredmeny:= a-b;
     '*' : eredmeny:= a*b;
   end;
   Muvelet:= eredmeny;
end;

procedure Kalkulator.Megjelenit;
begin
   writeln(a:5:2, muv_jel:2, b:5:2,' =', eredmeny:6:2);
end;

destructor Kalkulator.Done;
begin
   // Semmi teendőnk
end;

//  -----------  az UjKalkulator osztály  -----------
type
   UjKalkulator = object(Kalkulator)
     public
       constructor Init(a0,b0:real; muv_jel0:char);
       destructor Done;
       function Muvelet:real;
   end;

constructor UjKalkulator.Init(a0,b0:real; muv_jel0:char);
begin
   inherited Init(a0, b0, muv_jel0); // az ős konstruktorának hívása
end;

function UjKalkulator.Muvelet:real;
begin
   if muv_jel= '/' then
      if  b <> 0 then
         eredmeny := a / b
      else
         eredmeny := 0
   else
      eredmeny := inherited Muvelet;
   Muvelet:= eredmeny;
end;
```

```
destructor UjKalkulator.Done;
begin
   inherited; // az ős destruktorának hívása
   // Semmi teendőnk
end;

//  ----------  főprogramy  ----------
var
   x,y: real;
   m: char;
   k1  : Kalkulator;
   k2  : UjKalkulator;
begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+', '-' ,'*'];
   k1.Init(x,y,m);
   k1.Muvelet;
   k1.Megjelenit;
   k1.Done;
   writeln;
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*, /): '); readln(m);
   until m in ['+', '-' ,'*', '/'];
   k2.Init(x,y,m);
   k2.Muvelet;
   k2.Megjelenit;
   k2.Done;
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 3
2. adat: 4
műveleti jel (+,-,*): /
műveleti jel (+,-,*): *
 3.00 * 4.00 = 12.00

1. adat: 6
2. adat: 2
műveleti jel (+,-,*, /): /
 6.00 / 2.00 =  3.00
```

A futási eredményből látható, hogy a *Kalkulator* típusú objektumpéldány nem ismeri az osztás jelet, az *UjKalkulator* típusú objektumpéldány azonban elvégzi az osztást.

## 11.3 A sokalakúság

Az objektum-orientált programozás harmadik legfontosabb jellemzője a sokalakúság (*polymorphism*), amely lehetővé teszi a származtatott objektum viselkedésének (metódusainak) megváltoztatását. A sokalakúság csak akkor érvényesül, ha az eddig alkalmazott statikus metódusok egy része helyett ún. virtuális metódusokat használunk.

### 11.3.1 Az objektumok zártsága - statikus metódusok

Az *ujkalkprg* példaprogram készítésénél már találkoztunk azzal a problémával, hogy az ős metódusa nem hívja meg az utód újradefiniált metódusát. A példában az eredeti és az újradefiniált *Muvelet()* metódusok statikus metódusként jelennek meg, ami azt jelenti, hogy a fordító a hívásukat már a fordítás során rögzíti a kódban. Ennek megfelelően a *Kalkulator.Megjelenit()* eljárás mindig a *Kalkulator.Muvelet()* függvényt hívja, még akkor is, ha a *Megjelenit()* eljárást az *UjKalkulator* típusú *k2* objektumpéldány metódusaként hívjuk:

```
program ujkalzart;

// ...

procedure Kalkulator.Megjelenit;
begin
   writeln(a:5:2, muv_jel:2, b:5:2,' =', Muvelet:6:2);
end;

// ...

var
   x,y: real;
   m: char;
   k2  : UjKalkulator;
begin
   writeln;
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*, /): '); readln(m);
   until m in ['+', '-' ,'*', '/'];
   k2.Init(x,y,m);
   k2.Megjelenit;
   k2.Done;
   readln;
end.
```

A program futásának eredménye jól szemlélteti az elmondottakat:

```
1. adat: 144
2. adat: 12
műveleti jel (+,-,*, /): /
144.00 /12.00 =  0.00
```

A jelenség oka az objektumok zártsága, azaz a fordítási fázisban elvégzett korai kötés (*early binding*).

### 11.3.2 Virtuális metódusok

A virtuális metódusok alkalmazásával a program futása során dől el, hogy egy adott metódus melyik, újradefiniált változata hívódik. Ezt a jelenséget késői kötésnek (*late binding*) nevezzük.

A konstruktor kivételével minden metódus (a destruktor is) virtuálissá tehető a **virtual** módosító segítségével. A **virtual** módosítót az **object** struktúrában, a metódusok fejléce után kell megadni.

```
procedure pnév(paraméterlista); virtual;
function fnév(paraméterlista):típus; virtual;
destructor dnév(paraméterlista); virtual;
```

A virtuális metódust tartalmazó objektumtípusból származtatott típusokban is használnunk kell a **virtual** direktívát, ha az újradefiniált metódusváltozatra alkalmazni kívánjuk a késői kötést.

Más megközelítéssel elmondhatjuk, hogy a késői kötés lehetővé teszi, hogy az őstől örökölt, **virtual** szóval megjelölt metódust, az utódban új változattal helyettesítsük, vagyis lecseréljük. Ehhez az utódban az örökölt metódus fejlécével egyező fejlécű (újradefiniált) metódust kell létrehoznunk, és el kell látnunk a **virtual** módosítóval. (Amennyiben a két metódus paraméterezése különböző, a késői kötés helyett a túlterhelés mechanizmusa érvényesül!)

A virtuális metódusokat az objektumtípushoz készített, és futás közben feltöltött virtuális metódustábla (*VMT*) bejegyzési alapján hívja a futtató rendszer. A VMT feltöltése a konstruktor hívásakor megy végbe, így a virtuális metódusokat tartalmazó objektumokat mindig konstruktorral kell inicializálnunk. Ellenkező esetben futás közbeni hibát kapunk.

Készítsük el az *ujkalkprg* példaprogram virtuális metódusokat használó változatát! (*ujkalvirt*)

```
program ujkalvirt;

type
   Kalkulator = object
     protected
       a,b      : real;
       muv_jel  : char;
       eredmeny: real;
```

```
    public
      constructor Init(a0,b0:real; muv_jel0:char);
      function Muvelet:real; virtual;
      procedure Megjelenit;
  end;

constructor Kalkulator.Init(a0,b0:real; muv_jel0:char);
begin
  a       := a0;
  b       := b0;
  muv_jel := muv_jel0;
  eredmeny:= 0;
end;

function Kalkulator.Muvelet:real;
begin
  case muv_jel of
    '+' : eredmeny:= a+b;
    '-' : eredmeny:= a-b;
    '*' : eredmeny:= a*b;
  end;
  Muvelet:= eredmeny;
end;

procedure Kalkulator.Megjelenit;
begin
  writeln(a:5:2, muv_jel:2, b:5:2,' =', Muvelet:6:2);
end;

type
  UjKalkulator = object(Kalkulator)
    public
      constructor Init(a0,b0:real; muv_jel0:char);
      function Muvelet:real; virtual;
  end;

constructor UjKalkulator.Init(a0,b0:real; muv_jel0:char);
begin
  inherited Init(a0, b0, muv_jel0);
end;

function UjKalkulator.Muvelet:real;
begin
  if muv_jel= '/' then
     if  b <> 0 then
        eredmeny := a / b
     else
        eredmeny := 0
  else
     eredmeny := inherited Muvelet;
  Muvelet:= eredmeny;
end;

var
  x,y: real;
  m: char;
```

```
   k1  : Kalkulator;
   k2  : UjKalkulator;
begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+', '-' ,'*'];
   k1.Init(x,y,m);
   k1.Megjelenit;
   writeln;
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*, /): '); readln(m);
   until m in ['+', '-' ,'*', '/'];
   k2.Init(x,y,m);
   k2.Megjelenit;
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 4
2. adat: 5
műveleti jel (+,-,*): *
 4.00 * 5.00 = 20.00
1. adat: 6
2. adat: 2
műveleti jel (+,-,*, /): /
 6.00 / 2.00 =  3.00
```

### 11.3.3 Absztrakt metódusok

Az eddigi megoldásokkal, a virtuális metódusok lecserélése, csak lehetőségként jelent meg az objektumok továbbfejlesztői számára. Vannak esetek azonban, amikor a cserét kötelezővé kell tenni, a bizonytalan programállapotok elkerülése érdekében.

Az **abstract** módosítóval a virtuális metódusokat absztrakttá tehetjük. Az absztrakt metódust nem definiálja az objektumtípus, nincs törzse. Ennek következtében az **abstract** metódust tartalmazó osztály is absztrakttá válik, így nem készíthetünk vele objektumpéldányokat. Sőt az absztrakt, virtuális metódus lecserélése nélkül a származtatott osztály is absztrakt lesz.

Az elmondottak figyelembevételével a *Kalkulator* ← *UjKalkulator* objektum-hierarchiánkat *Alap* ← *Kalkulator* ← *UjKalkulator* alakúvá formálhatjuk. Az absztrakt *Alap* objektumtípus csak a *Muvelet()* absztrakt virtuális metódust deklarálja:

```
type
   Alap = object
     public
       function Muvelet:real; virtual; abstract;
   end;

   Kalkulator = object (Alap)
     protected
       a,b      : real;
       muv_jel  : char;
       eredmeny: real;
     public
       constructor Init(a0,b0:real; muv_jel0:char);
       function Muvelet:real; virtual;
       procedure Megjelenit;
   end;

   UjKalkulator = object(Kalkulator)
     public
       constructor Init(a0,b0:real; muv_jel0:char);
       function Muvelet:real; virtual;
   end;
```

# 12. Az Object Pascal mód további lehetőségei

Könyvünk utolsó fejezetében azokat a nyelvi elemeket kötjük csokorba, amelyek az ***Object Free Pascal*** nyelv részei. A bemutatott megoldások többsége ***ObjFPC*** és ***Delphi*** fordítási módban egyaránt elérhető, azonban vannak kivételek is, amiket az eddigiekhez hasonlóan jelzünk. Az ***Object Free Pascal*** olyan módon bővíti a Pascal nyelvet, hogy rá építve grafikus felhasználói felülettel rendelkező alkalmazásokat lehet fejleszteni. Mint ahogy a bevezetőben említettük, erre önmagában a ***Free Pascal*** fordító nem alkalmas, azonban a nyelv ismeretét jól kamatoztathatjuk a *Borland Delphi* vagy a *Lazarus* környezetben történő programozás során.

## 12.1 Globális tulajdonságok - a property deklaráció

Nagyobb alkalmazások fejlesztése során gyakran előfordul, hogy a unitokban elhelyezett globális változókat megosztva használják a program moduljai. Az ilyen változók közvetlen módosítását semmilyen formában sem tudjuk nyomon követni, ellenőrizni, ami nehezen felderíthető programhibákhoz vezethet.

Ha az alábbi példában, a *Masiku* modul valamelyik alprogramja az *x* változó értékét negatívra állítja, a program futása hibajelzéssel megszakad:

```
program Foprog;
uses Globalu, Masiku;
begin
  // x-et módosító hívások
  writeln(sqrt(x):8:3);
  readln;
end.
```

```
unit Globalu;

interface
  var
    x : real;
implementation

begin
  x:= 12;
end.
```

Az ilyen és hasonló problémák megoldására, biztosít megoldást ***ObjFPC*** módban a globális tulajdonság (**property**):

```
{$mode ObjFPC}
unit Globalu;

interface
  function LekerX : real;
  procedure BeallitX(a : real);
  property x : real read LekerX write BeallitX;

implementation
  var
    xx : real; // itt tároljuk a tulajdonság értékét
```

```
  // csak nem negatív értéket fogadunk el
  procedure BeallitX(a : real);
  begin
    if a>=0 then xx := a;
  end;

  // itt nincs szükség ellenőrzésre
  function LekerX : real;
  begin
    LekerX := xx;
  end;

begin
  xx := 12;
end.
```

A példában az *x* tulajdonság csupán egy ***real*** típusú nevet jelöl, amelyre vonatkozó értékadás során a **write** után megadott eljárás, olvasásakor pedig a **read** után álló függvény hívódik meg. Az elmondottakat jól szemlélteti a főprogram két változata: bal oldalon az *x* tulajdonságot használjuk, jobb oldalon pedig azt a kódot, amivé a fordító lefordítja a tulajdonság elérését:

```
program Foprog;
uses Globalu;

begin
  x := 144;
  writeln(sqrt(x):8:3);
  x := -144;
  writeln(sqrt(x):8:3);
  readln;
end.
```

```
program Foprog;
uses Globalu;

begin
  BeallitX(144);;
  writeln(sqrt(LekerX):8:3);
  BeallitX(-144);
  writeln(sqrt(LekerX):8:3);
  readln;
end.
```

A beállító (*set*) eljárás egyetlen paraméterének, valamint a lekérdező (*get*) függvény értékének a tulajdonsággal megegyező típusúnak kell lennie. A globális elérési alprogramokat az alkalmazás bármelyik moduljában elhelyezhetjük.

A fenti példában szereplő *x* tulajdonság írható és olvasható. Amennyiben a **property** sorból elhagyjuk a **read** részt csak írható, míg a **write** rész elhagyásával csak olvasható tulajdonsághoz jutunk:

```
property xrw : real read LekerX write BeallitX;
property xwo : real write BeallitX;
property xro : real read LekerX;
```

A tulajdonság többértékű is lehet, amely értékek közül egy index segítségével választhatjuk ki a szükséges elemet. Az alábbi *Lottou* unitban egy lottószelvényt kezelünk a *Lotto* indexelt tulajdonság segítségével. Szükség esetén olyan tulajdonságot is létrehozhatunk, amelynél az index értékét előre rögzítjük (*Lotto5*).

```
{$mode ObjFPC}
unit Lottou;

interface
  function GetLotto(n:integer):byte;
  procedure SetLotto(n:integer; szam:byte);
  property Lotto[n:integer] : byte read GetLotto write SetLotto;
  property Lotto5 : byte index 5 read GetLotto write SetLotto;

implementation
  var
    mlotto : array[1..5] of byte;

  function GetLotto(n:integer):byte;
  begin
    GetLotto := mlotto[n];
  end;

  procedure SetLotto(n:integer; szam:byte);
  begin
    mlotto[n] := szam;
  end;
end.
```

A *Lottou* unitot használó főprogram „kitölti a lottószelvényt", majd pedig ellenőrzi az első és az ötödik számot:

```
program Lottozo;
uses Lottou;
var i:integer;
begin
  randomize;
  for i:=1 to 5 do
     Lotto[i] := random(90)+1;
  writeln(Lotto[1]:4);
  writeln(Lotto[5]:4, Lotto5:4);
  readln;
end.
```

Az indexelt tulajdonság, csak első látásra hasonlít az egydimenziós tömbökre, mivel ebben az esetben az index értéke akár **string** típusú is lehet:

```
{$mode ObjFPC}
unit Szotaru;

interface
  function GetSzo(si : string) : string;
  procedure SetSzo(si : string; s : string);
  property Szotar[si:string] : string read GetSzo write SetSzo;

implementation

  var
    NemIgen : array[0..1] of string;
```

```
  function GetSzo(si : string) : string;
  begin
    if Upcase(si)='IGEN' then
      GetSzo := NemIgen[1]
    else if Upcase(si)='NEM' then
      GetSzo := NemIgen[0]
    else
      GetSzo := '';
  end;

  procedure SetSzo(si : string; s : string);
  begin
    if Upcase(si)='IGEN' then
        NemIgen[1] := s
    else if Upcase(si)='NEM' then
        NemIgen[0] := s
  end;

begin
  NemIgen[0] := 'No';
  NemIgen[1] := 'Yes';
end.
```

A *Szotaru* unitban definiált *Szotar* tulajdonság elérése során, igen érdekes szövegkörnyezetben használjuk a szögletes zárójeleket:

```
program SzotarPrg;
uses Szotaru;
begin
  writeln(Szotar['Igen']);    // Yes
  writeln(Szotar['Nem']);     // No
  Szotar['Nem']:='Nein';
  writeln(Szotar['Nem']);     // Nein
  readln;
end.
```

A tulajdonságok felhasználásának igazi területe a **class** típusú osztályokra épülő, objektum-orientált programozás, ahol a publikált (*published*) tulajdonságok a vizuális fejlesztőrendszerekből is elérhetők.

## 12.2 Objektumosztályok - a class típus

A 11. fejezetben megismerkedtünk az objektum-orientált programozás alapelveivel és az **object** típussal. Az **object** típusú objektumpéldányok használatának hátránya, hogy csak a program futása során érhetők el. A vizuális fejlesztőrendszerek igényeinek megfelelően újratervezték az *Object Pascal* nyelv objektum-orientált lehetőségeit és objektummodelljét. A kialakított új, **class** típusú programegység (komponens, építőelem), olyan dinamikus objektum, amely fejlesztési és futási időben egyaránt felhasználható.

### 12.2.1 Az objektumosztályok (class) felépítése

A **class** típus felépítése sokban emlékeztet az **object** típus szerkezetére, amelyet ismertnek tételezünk fel. Alapvetően, az osztályok is adatokból és rajtuk műveletet végző metódusból állnak. Újdonság azonban a tulajdonságok megjelenése az osztálytípusban.

```
type
   osztálynév = ⟨packed⟩ class(ősosztály)
      mező1: típus1;
    ...
      mezőL: típusN;
      metódus1 fejléce;
    ...
      metódusM fejléce;
      tulajdonság1;
    ...
      tulajdonságN;
   end;

Metódusdefiníciók
```

Az osztályok egyetlen hierarchiába szerveződnek, melynek kiindulópontja, gyökere a ***System*** unitban definiált ***TObject*** osztály. Ha a **class** szó után nem adjuk meg az *ősosztályt*, akkor az automatikusan a ***TObject*** lesz, így minden új osztály rendelkezik annak minden nyilvános metódusával. Ezek közül a legfontosabbak a ***Create()*** konstruktor, a ***Destroy()*** virtuális destruktor és a ***Free()*** metódus, amely csak akkor hívja meg a destruktort, ha a **self** mutató nem **nil** értékű.

A metódusok között megjelentek az ún. osztálymetódusok (*class methods*), amelyek nem az objektumpéldányokhoz, hanem magához az osztályhoz tartoznak. Az osztálymetódusok definíciója a **class** szóval kezdődik, és nem rendelkeznek a **self** paraméterrel, ezért belőlük nem érhetjük el sem az adatmezőket, sem a normál metódusokat, sem pedig a tulajdonságokat.

```
{$mode ObjFPC}
type
  Osztaly = class
    class function Osztalynev : string;
  end;

class function Osztaly.Osztalynev : string;
begin
  result := 'Osztaly';
end;

begin
  writeln(Osztaly.Osztalynev);
  readln;
end.
```

Természetesen a normál eljárás, függvény, konstruktor és destruktor metódusok rendelkeznek a **self** paraméterrel, amely osztályok esetén egy mutató (referencia).

Az objektum-orientált programozás egyik fontos szabálya, hogy ne engedjük meg az objektum adatmezőinek közvetlen, ellenőrzés nélküli elérését. A biztonságos adatelérést metódusok hívásával, illetve tulajdonságok bevezetésével valósíthatjuk meg. A metódusok hívása kissé körülményes, és rontja a program olvashatóságát. Ezzel szemben tulajdonságok használatával a program olvashatósága megmarad. A tulajdonságok megadásának általános alakja:

```
property tulajdonságnév : típus read  adatmező | metódus
                                write adatmező | metódus;
```

Az osztályokban definiált tulajdonságokra ***Delphi*** módban is érvényesek a 12.1. részben elmondottak. A különbség csak annyi, hogy a **read** és a **write** szekciókban a metódusnév helyett adatmezőnevet is megadhatunk.

```
type  TPont = class
       private
           // adatmezők
           fx,fy : integer;
       public
           constructor Letrehoz(a,b : integer);
           destructor Megszuntet;
           // elérési metódusok
           function  LekerX : integer;
           procedure BeallitX(a : integer);
           function  LekerY : integer;
           procedure BeallitY(a : integer);
           // írható/olvasható tulajdonságok
           property x  : integer read LekerX write BeallitX;
           property y  : integer read fx  write fy;
      end;
```

Megjegyezzük, hogy a példával ellentétben, az osztályokban definiált konstruktorok szokásos neve a *Create*, míg a destruktorokat *Destroy* néven hozzuk létre. A névválasztás utal az általuk elvégzett tevékenységekre.

A osztályelemek elérésének szabályozása, a **private**, **protected** és **public** foglalt szavakkal, az **object** típusnál megismert módon történik. Újdonságot jelent a **published** (publikált) alapszó használata, amelyre a **public**-kal azonos elérési szabályok vonatkoznak. Egyetlen eltérés a kétféle nyilvános elérési mód között az, hogy a **published** részben elhelyezkedő osztálytípusú adatmezőkhöz, metódusokhoz és tulajdonságokhoz futásidejű típus-információkat (*runtime type information - RTTI*) rendel a fordító. Ezzel a megoldással ismeretlen típusú objektumok publikált elemei is felderíthetők. Megjegyezzük, hogy a futásidejű típus-információk előállítását a {*$M+*} vagy a {*$TYPEINFO ON*} fordítási direktívával kell kérnünk a fordítótól.

Az osztályon belül a **private**, a **public**, a **protected** és a **published** részek tetszőleges sorrendben és számban elhelyezkedhetnek, azonban minden egyes részben a metódusok és a tulajdonságok csak az adatmezők után következhetnek. Az adatrejtés elvének érvényesüléséhez az osztálytípust is unit **interface** részébe kell helyeznünk.

Az osztálydeklaráció csak a **program**, a **library** és a **unit** (**interface** vagy **implementation**) deklarációs részében adható meg. Eljárás és függvény deklarációs része nem tartalmazhat osztálytípust.

Az elmondottak alapján készítsük el a 11. fejezet *Kalkulator* objektumtípusának osztályváltozatát *TKalkulator* néven, egy *Kalkulator* nevű unitban, kiegészítve az operandusok és a műveleti jel elérését segítő tulajdonságokkal! (*Kalkulator*)

```
{$mode ObjFPC}
unit Kalkulator;

interface

type
   TKalkulator = class
     private
       adat1, adat2, eredmeny: real;
       muv_jel : char;
     public
       constructor Create(a0,b0:real; muv_jel0:char);
       destructor Destroy;
       procedure Megjelenit;
       function Muvelet:real;
       procedure Set_muvjel(muv_jel0 : char);
       property a : real read adat1 write adat1;
       property b : real read adat2 write adat2;
       property c : real read Muvelet;
       property op: char read muv_jel write Set_muvjel;
   end;

implementation

constructor TKalkulator.Create(a0,b0:real; muv_jel0:char);
begin
   adat1    := a0;
   adat2    := b0;
   muv_jel  := muv_jel0;
   eredmeny:= 0;
end;

destructor TKalkulator.Destroy;
begin
end;

procedure TKalkulator.Megjelenit;
begin
   writeln(adat1:5:2, muv_jel:2, adat2:5:2,' =', Muvelet:6:2);
end;
```

```
function TKalkulator.Muvelet:real;
begin
   case muv_jel of
      '+' : eredmeny:= adat1+adat2;
      '-' : eredmeny:= adat1-adat2;
      '*' : eredmeny:= adat1*adat2;
   end;
   result := eredmeny;
end;

procedure TKalkulator.Set_muvjel(muv_jel0 : char);
begin
   if muv_jel0 in ['+','-','*'] then
      muv_jel:= muv_jel0
   else
      muv_jel:='+';
end;

end.
```

### 12.2.2 Objektumok létrehozása

Bár a **class** és az **object** típusok felépítése között sok a hasonlóság, alapvetően különbözik egymástól a kétféle objektummodell. Az **object** típusú objektumpéldányokat a Pascal nyelv, változókra vonatkozó szabályai szerint hozzuk létre. Ezzel szemben az osztályobjektumokat az objektum-referencia modell szerint kell kezelnünk.

Első lépés a változódeklaráció, melynek során egy rejtett mutató ún. objektumreferencia jön létre, amelyet **nil** értékkel inicializálhatunk is:

```
var
   k1 : TKalkulator = nil;
```

A dinamikus objektumpéldányok a konstruktorok osztálymetódusként való hívásával keletkeznek. Amennyiben az osztályunk nem tartalmaz saját konstruktort, a ***TObject***-től örökölt ***Create()*** konstruktort kell használnunk:

```
begin
   k1 := TKalkulator.Create(12,23,'+');
end.
```

A *k1* ezek után az objektumpéldányra hivatkozik (mutat). Ennek ellenére, a *k1* mutatóval adatmezőre, tulajdonságra vagy metódusra történő hivatkozás során nem használhatjuk a *k1^.valami* formát. A bonyolult jelölésrendszer elkerülése céljából a *k1. valami* kifejezés helyes értelmezése a fordítóprogram feladata. Természetesen a **with** utasítást is alkalmazhatjuk:

```
begin
   k1 := TKalkulator.Create(12,23,'+');
   with k do Megjelenit;
```

```
        k.Create(k.a+12, k.b-23,'*');
        k.Megjelenit;
        k.Destroy;
    end.
```

A példában a *Create()* konstruktort normál metódusként is hívjuk. Ekkor feladata csupán az objektumpéldány inicializálása, ugyanúgy, mint az **object** objektumtípusok esetén.

Ha már nincs többé szükségünk az objektumpéldányra, a destruktor hívásával törölhetjük azt a memóriából. Saját destruktor hiányában, a ***TObject*** osztálytól örökölt *Destroy()* destruktort, vagy a *Free()* metódust szintén alkalmazhatjuk.

Készítsünk főprogramot a *Kalkulator* unitban tárolt *TKalkulator* osztály használatának bemutatására! (*Kalkprog*)

```
program Kalkprog;

uses Kalkulator;

var
   x,y: real;
   m: char;
   k : TKalkulator;

begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*): '); readln(m);
   until m in ['+','-','*'];

   k:=TKalkulator.Create(x,y,m);
   with k do
      Megjelenit;

   k.Create(k.a*2, k.b*3,'*');
   k.Megjelenit;

   k.a:=2;
   k.op:='-';
   k.Megjelenit;
   k.Destroy;
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 11
2. adat: 12
műveleti jel (+,-,*): +
11.00 +12.00 = 23.00
22.00 *36.00 =792.00
 2.00 -36.00 =-34.00
```

Felhívjuk a figyelmet, hogy az azonos osztályú objektumpéldányok sem másolhatók az értékadó utasítás segítségével. Ennek oka, hogy az értékadás során csupán az objektumpéldányok referenciája másolódik, így a művelet után mindkét változó ugyanarra az objektumra hivatkozik:

```
uses Kalkulator;
var
  k1, k2 : TKalkulator;
begin
  k1:=TKalkulator.Create(12, 3, '*');
  k1.Megjelenit;                        // 12.00 * 3.00 = 36.00
  k2 := k1;
  k2.a  := 12;
  k2.b  := 23;
  k2.op := '+';
  k1.Megjelenit;                        // 12.00 +23.00 = 35.00
  k1.Destroy;
end.
```

### 12.2.3 Öröklés és a sokalakúság

Az öröklés fogalmai és eszközei alapvetően megegyeznek a **class** és az **object** típusok esetén:

- az egyetlen ős t az alapszó után, kerek zárójelben adjuk meg, az utód típusdefiníciójában,
- az utód metódusaiból a közvetlen ős metódusait az **inherited** alapszó segítségével hívhatjuk,
- az utód típusú objektumpéldány mindig helyettesítheti az ős objektumpéldányát.

Alapvető eltérés a két objektumtípus között, hogy a **class** típusok mindig származtatással jönnek létre, még akkor is, ha ezt nem jelöljük az osztálydefinícióban. Alapértelmezés szerint ekkor az ősosztály a ***TObject***, amelytől minden osztály egy sor metódust örököl. Az elmondottakat egy egyszerű példával szemléltetjük:

```
{$mode ObjFPC}
program orokles;
type                        // Az ősosztály definíciója
    TOs = class             // vagy class(TObject)
        private
          a : integer;
        public
          constructor Create(a0 : integer);
          procedure Kiir;
    end;

constructor TOs.Create(a0 : integer);
begin
  a := a0;
end;
```

```
procedure TOs.Kiir;
begin
  writeln('a= ', a);
end;

type                          // A származtatott osztály definíciója
    TUtod = class(TOs)
        private
          b : integer;
        public
          constructor Create(a0 : integer; b0 : integer);
          procedure Kiir;
    end;

constructor TUtod.Create(a0 : integer; b0 : integer);
begin
  inherited Create(a0);
  b := b0;
end;

procedure TUtod.Kiir;
begin
  inherited;
  writeln('b= ', b);
end;

var
  x : TOs;
  y : TUtod;
begin
  x:=TOs.Create(7);           // TOs objektum létrehozása
  x.Kiir;                     // a= 7

  y:=TUtod.Create(12,23);     // TUtod objektum létrehozása
  y.Kiir;                     // a= 12 / b= 23
  x.Free;                     // TOs objektum megszüntetése

  x := y;
  x.Kiir;                     // a= 12
  y.Free;                     // TUtod objektum megszüntetése
  readln;
end.
```

Az *x:=y;* értékadás során csak az utód hivatkozása másolódik az ős hivatkozásába, míg a létrehozott két objektumpéldány változatlan marad. Az értékadás előtt az *x* által korábban hivatkozott objektumot meg kell szüntetnünk, hisz az értékadás után elérhetetlen lesz számunkra.

A sokalakúság megvalósítása szintén sok hasonlóságot mutat a kétféle objektumtípus között. Ugyanúgy beszélhetünk a statikus metódusok korai kötéséről, illetve a virtuális metódusok késői kapcsolódásáról, valamint az absztrakt metódusokról. Vannak azonban lényeges különbségek is, amelyekre külön ki kell térnünk.

Jól ismert számunkra, hogy a sokalakúság az örökölt virtuális metódusok teljes lecserélésével valósul meg. A cserélhetőséget az osztálydefinícióban, a metódus fejléce mögött elhelyezett **virtual** módosítóval jelezzük. Valamely metódus az osztály-hierarchia tetszőleges pontján (osztályában) virtuálissá tehető. A továbbiakban minden ebből származtatott osztályban használnunk kell az **override** módosítót, ha a virtuális metódust le kívánjuk cserélni, hatálytalanítva az örökölt változatot.

Az öröklést bemutató példaprogramunk módosított változatában a *toString()* (karaktersorozattá alakító) metódusok cseréjét (újradefiniálását) valósítottuk meg. Érdemes megfigyelni, hogy a csere akkor is érvényesül, ha őstípusú változóval hivatkozunk az utód objektumára (a főprogram utolsó *x.Kiir;* utasítása).

```
{$mode ObjFPC}
program sokalakusag;
uses SysUtils;

type
     TOs = class(TObject)
        private
          a : integer;
        public
          constructor Create(a0 : integer);
          procedure Kiir;
          function toString : String; virtual;
     end;

constructor TOs.Create(a0 : integer);
begin
  a := a0;
end;

procedure TOs.Kiir;
begin
  writeln(toString);
end;

function TOs.toString:string;
begin
  result := 'a = ' + inttostr(a);
end;

type
     TUtod = class(TOs)
        private
          b : integer;
        public
          constructor Create(a0 : integer; b0 : integer);
          function toString : String; override;
     end;
```

```
constructor TUtod.Create(a0 : integer; b0 : integer);
begin
  inherited Create(a0);
  b := b0;
end;

function TUtod.toString:string;
begin
  result := inherited tostring + ' / b = ' + inttostr(b);
end;

var
  x : TOs;
  y : TUtod;
begin
  x:=TOs.Create(7);
  x.Kiir;                 // a= 7
  y:=TUtod.Create(12,23);
  y.Kiir;                 // a= 12 / b= 23
  x.Free;
  x := y;
  x.Kiir;                 // a= 12 / b= 23
  y.Free;
  readln;
end.
```

Amennyiben véletlenül az **override** módosító helyett a **virtual** szót használjuk (mint az **object** típusok esetén), akkor egészen más eredményt kapunk:

```
a = 7
a = 12
a = 12
```

Ekkor a csere helyett újradeklaráltuk a ***toString()*** függvényt, elfedve az örökölt virtuális metódust, amely ezáltal statikussá vált.

Származtassunk *TKalkulator* osztályból egy *TUjKalkulator* típust, amely az osztás operátorával bővíti a műveleteket! Készítsünk az osztályok használatát bemutató főprogramot is! (*UjKalkprog*)

A *TUjKalkulator* típust szintén a *Kalkulator* unitban helyezzük el. Az eredeti osztályunkat is módosítani kell, az adatmezők elérésének védetté változtatásával, illetve a *Muvelet()* és *Set_muvjel()* metódusok virtuálissá tételével.

```
{$mode ObjFPC}
unit Kalkulator;

interface

type
   TKalkulator = class
     protected
       adat1, adat2, eredmeny: real;
       muv_jel : char;
```

```
    public
      constructor Create(a0,b0:real; muv_jel0:char);
      procedure Megjelenit;
      function Muvelet:real; virtual;
      procedure Set_muvjel(muv_jel0 : char); virtual;
      property a : real read adat1 write adat1;
      property b : real read adat2 write adat2;
      property c : real read Muvelet;
      property op: char read muv_jel write Set_muvjel;
  end;

type
  TUjKalkulator = class(TKalkulator)
    public
      constructor Create(a0,b0:real; muv_jel0:char);
      function Muvelet:real; override;
      procedure Set_muvjel(muv_jel0 : char); override;
  end;

implementation

constructor TKalkulator.Create(a0,b0:real; muv_jel0:char);
begin
  adat1   := a0;
  adat2   := b0;
  muv_jel := muv_jel0;
  eredmeny:= 0;
end;

procedure TKalkulator.Megjelenit;
begin
  writeln(adat1:5:2, muv_jel:2, adat2:5:2,' =', Muvelet:6:2);
end;

function TKalkulator.Muvelet:real;
begin
  case muv_jel of
    '+' : eredmeny:= adat1+adat2;
    '-' : eredmeny:= adat1-adat2;
    '*' : eredmeny:= adat1*adat2;
  end;
  result := eredmeny;
end;

procedure TKalkulator.Set_muvjel(muv_jel0 : char);
begin
  if muv_jel0 in ['+','-','*'] then
    muv_jel:= muv_jel0
  else
    muv_jel:='+';
end;

constructor TUjKalkulator.Create(a0,b0:real; muv_jel0:char);
begin
  inherited Create(a0, b0, muv_jel0);
end;
```

```
function TUjKalkulator.Muvelet:real;
begin
   if muv_jel= '/' then
      if  b <> 0 then
         eredmeny := a / b
      else
         eredmeny := 0
   else
      eredmeny := inherited Muvelet;
   result := eredmeny;
end;

procedure TUjKalkulator.Set_muvjel(muv_jel0 : char);
begin
   if muv_jel0 = '/' then
      muv_jel:= muv_jel0
   else
      inherited Set_muvjel(muv_jel0);
end;
end.

program UjKalkprog;
uses Kalkulator;
var
   x,y: real;
   m: char;
   k : TUjKalkulator;
begin
   write('1. adat: '); readln(x);
   write('2. adat: '); readln(y);
   repeat
      write('műveleti jel (+,-,*,/): '); readln(m);
   until m in ['+','-','*', '/'];

   k:=TUjKalkulator.Create(x,y,m);
   with k do
      Megjelenit;

   k.Create(k.a+2, k.b-3,'+');
   k.Megjelenit;

   k.b:=2;
   k.op:='/';
   k.Megjelenit;
   k.Free;
   readln;
end.
```

*A program futásának eredménye:*

```
1. adat: 276
2. adat: 12
műveleti jel (+,-,*,/): /
276.00 /12.00 = 23.00
278.00 + 9.00 =287.00
278.00 / 2.00 =139.00
```

### 12.2.4 Osztályműveletek: is, as

Az Object Pascal két olyan műveletet is definiál (**is**, **as**), amelyek operandusai objektum-, illetve osztály-hivatkozások.

Az **is** operátorral dinamikus típusellenőrzést végezhetünk. Megtudhatjuk, hogy egy adott objektum típusa megfelel-e a megadott osztálynak, vagy sem. Az **as** operátort pedig típuskonverzió kijelölésére használjuk.

```
{$mode ObjFPC}
program is_as;
uses Kalkulator;

procedure Megjelenit(p:TKalkulator);
begin
  if p is TUjKalkulator then
     with (p as TUjKalkulator) do
       writeln('ÚjKalkulátor : ', a:6:2,op:2,b:6:2,'=',c:8:2)
  else
  if p is TKalkulator then
     with (p as TKalkulator) do
       writeln('Kalkulátor   : ', a:6:2,op:2,b:6:2,'=',c:8:2)
end;

var
  k1 : TKalkulator;
  k2 : TUjKalkulator;
begin
  k1 := TKalkulator.Create(12, 23, '+');
  Megjelenit(k1);
  k1.Free;
  k2 := TUjKalkulator.Create(144, 12, '/');
  Megjelenit(k2);
  k2.Free;
  readln;
end.
```

A fenti példában, a *Megjelenit()* függvényben felderítjük a hivatkozásával átadott objektum típusát, majd pedig ennek megfelelően kiírjuk a nyilvános tulajdonságainak értékét:

```
Kalkulátor   :  12.00 + 23.00=   35.00
ÚjKalkulátor : 144.00 / 12.00=   12.00
```

## 12.3 Objektum-interfészek - az interface típus

Mint említettük, a Free Pascal nem támogatja a több ősosztálytól való származtatást, a többszörös öröklést. Bizonyos esetekben azonban az osztály kapcsolódási felületének kialakításakor olyan metódusok megvalósítását is elő kell írni, amelyek nem szerepelnek az absztrakt ősosztály-hierarchiában. A megvalósítandó metódusok deklarációját

csokorba gyűjthetjük, és interfészként az osztályhoz kapcsolhatjuk. Az osztályok származtatási listájában egyetlen ősosztály és tetszőleges számú interfész szerepelhet.

```
type
   osztálynév = class (ősosztálynév, interfész1, interfész2, ...)
      // ...
   end;
```

Bizonyos metódusok kötelező megvalósításának igénye elsősorban az operációs rendszerek szintjén megvalósított, objektumalapú technológiákkal (*COM*, *CORBA* stb.) együtt jelent meg. A megoldás hatásosságát jelzi, hogy a *Java* nyelvben széles körben alkalmazzák. Az interfésztípus általános felépítése:

```
type
   interfésznév = interface(ősinterfész1, ősinterfész2, ...)
      ['{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}']    // GUID
      metódusfejléc1;
      // ...
      metódusfejlécM;
      tulajdonság1;
      // ...
      tulajdonságN;
   end;
```

A globális egyedi azonosítót (*GUID*) csak Windows alatt használjuk, a COM modellhez való kapcsolódás során.

Az interfészek használatának jobb megértése érdekében készítsünk egy *IAzonositas* interfészt, amely gyártóhoz rendeli, és sorozatszámmal látja el a kalkulátorainkat! (*Interfeszprog*)

A példában az azonosítást biztosító interfész esetén nem adunk meg ősinterfészt, így az automatikusan a ***System*** unitban definiált ***IUnknown*** interfésztől származik.

```
type
   IAzonositas = Interface
      function Modell : string;
      function Sorozatszam : string;
   end;
```

Az ***IUnknown*** interfészben deklarált metódusok definiálásával nem kell bajlódnunk, ha a *TKalkulator* osztályt a ***TInterfacedObject*** osztályból származtatjuk (melynek ősei a ***TObject*** és az ***IUnknown***), amely az interfészek használatához szükséges alapmetódusokat is tartalmazza.

Ennek megfelelően módosítanunk kell a *TKalkulator* osztály típusát a *Kalkulator* unitban:

```
type
   TKalkulator = class(TInterfacedObject)
     protected
       adat1, adat2, eredmeny: real;
       muv_jel : char;
     public
       constructor Create(a0,b0:real; muv_jel0:char);
       procedure Megjelenit;
       function Muvelet:real; virtual;
       procedure Set_muvjel(muv_jel0 : char); virtual;
       property a : real read adat1 write adat1;
       property b : real read adat2 write adat2;
       property c : real read Muvelet;
       property op: char read muv_jel write Set_muvjel;
   end;
```

A feladat megoldásából jól látható, hogyan használtuk az *IAzonositas* interfészt az azonosításra:

```
{$mode ObjFPC}
program Interfeszprog;

uses Kalkulator;

type
   IAzonositas = Interface
      function Modell : string;
      function Sorozatszam : string;
   end;

// HP 12c Platinum kalkulátor
type
   THPKalkulator = class(TUjKalkulator, IAzonositas)
       private
          ssz : string;
       public
          function Modell : string;          // az interfészből
          function Sorozatszam : string;     // az interfészből
          constructor Create(sszam:string);
   end;

constructor THPKalkulator.Create(sszam:string);
begin
  ssz := sszam;
  inherited Create(1,1,'+');
end;

function THPKalkulator.Modell : string;
begin
  result:='HP 12c Platinum';
end;

function THPKalkulator.Sorozatszam : string;
begin
  result:=ssz;
end;
```

```
// Texas TI-89 Titanium Programmable Graphing Calculator
type
   TTexasKalkulator = class(TUjKalkulator, IAzonositas)
       private
          ssz : string;
       public
          function Modell : string;          // az interfészből
          function Sorozatszam : string;     // az interfészből
          constructor Create(sszam:string);
   end;

constructor TTexasKalkulator.Create(sszam:string);
begin
  ssz := sszam;
  inherited Create(1,1,'+');
end;

function TTexasKalkulator.Modell : string;
begin
  result:='Texas TI-89 Titanium';
end;

function TTexasKalkulator.Sorozatszam : string;
begin
  result:=ssz;
end;

// Főprogram
var
   hp1, hp2 : THPKalkulator;
   texas    : TTexasKalkulator;

begin
   hp1:=THPKalkulator.Create('12-LN-23-1979');
   hp2:=THPKalkulator.Create('23-LN-12-2003');
   texas := TTexasKalkulator.create('79-LL-1223');
   writeln(hp1.Modell,#9#9,hp1.Sorozatszam);
   writeln(hp2.Modell,#9#9,hp2.Sorozatszam);
   writeln(texas.Modell,#9,texas.Sorozatszam);
   hp1.Megjelenit;
   hp1.a := 12;
   hp1.b := 23;
   hp1.op:='+';
   hp1.Megjelenit;
   hp1.free;
   hp2.free;
   texas.free;
   readln;
end.
```

*A program futásának eredménye:*

```
HP 12c Platinum          12-LN-23-1979
HP 12c Platinum          23-LN-12-2003
Texas TI-89 Titanium     79-LL-1223
 1.00 + 1.00 =  2.00
12.00 +23.00 = 35.00
```

## 12.4 Kivételek kezelése

A kivétel (*exception*) olyan hibás állapot vagy esemény, amely megszakítja az alkalmazás szabályszerű futását. Amikor fellép egy kivétel, a vezérlés a futó program aktuális pontjáról a kivételkezelőhöz (*exception handler*) kerül.

A kivételek megvalósítására osztályobjektumokat használunk, aminek legfontosabb előnyei:

- a kivételeket, az öröklés felhasználásával, hierarchiákba lehet csoportosítani,
- újabb kivételeket definiálhatunk, a meglevő kód megváltoztatása nélkül,
- kivételobjektum információkat (például hibaüzenet vagy hibakód) vihet a keletkezés helyéről a kezelési pontra.

A kivételkezelés alkalmazásához, a programunkhoz kell kapcsolni ***SysUtils*** unitot, amely a futásidejű könyvtárhoz (*run-time library*) definiálja a kivételeket. Amikor egy alkalmazás használja a ***SysUtils*** modult, a futásidejű hibák automatikusan kivételekké alakulnak. Ennek köszönhetően a nullával való osztást, vagy az általános védelmi hibát stb., amelyek máskülönben az alkalmazás leállásához vezetnének, a programunk adott pontján „elfoghatjuk", és strukturált módon kezelhetjük.

### 12.4.1 Kivételek deklarációja

Object Pascal-ban a kivételek osztályok, így a kivétel deklarációjához a szokásos osztálydeklarációs megoldásokat használhatjuk. A kivételobjektum (*exception object*) tetszőleges osztály példánya lehet, azonban ajánlatos minden kivételt a ***SysUtils*** modulban definiált ***Exception*** osztályból származtatni:

```
type
  EMathError   = class(EExternal);
  EInvalidOp   = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow    = class(EMathError);
  EUnderflow   = class(EMathError);
```

Az egymással összefüggő kivételeket – az öröklés adta lehetőségeket kihasználva – családokba csoportosíthatjuk. A példában szereplő deklarációk a matematikai kivételek családját definiálják. A kivételcsaládok révén lehetőség nyílik arra, hogy a kivételek adott csoportját egyetlen névvel azonosítsuk. Például, az ***EMathError*** kivételt feldolgozó kód kezeli az összes ***EMathError*** osztályból közvetlenül vagy közvetett módon származtatott kivételt.

A származtatott kivételosztályokban újabb adatmezőket, metódusokat és tulajdonságokat vezethetünk be, a kivételhez kapcsolódó kiegészítő információk átadására. Pél-

dául, a ***SysUtils*** unitban definiált ***EInOutError*** osztály az *ErrorCode* nyilvános elérésű adatmezőben továbbítja a kivételt kiváltó állományhiba kódját:

```
type
  EInOutError = class(Exception)
    public
      ErrorCode: LongInt;
  end;
```

A ***SysUtils*** unitban tárolt ***Exception*** osztály nyilvános részének legfontosabb deklarációi:

```
type
  Exception = class(TObject)
   public
    constructor Create(const msg : string);
    constructor CreateFmt(const msg : string;
                          const args : array of const);
    constructor CreateRes(ResString: PString);
    constructor CreateResFmt(ResString: PString;
                             const args: array of const);
    property message: string read fmessage write fmessage;
    // ....
end;
```

Az ***Exception*** osztály *message* tulajdonságát minden kivételosztály örökli. Ennek köszönhetően a kivételobjektumok rendelkeznek a kivételállapotot leíró üzenettel. Az osztály konstruktorai különböző metódusokat ajánlanak fel a *message* tulajdonság inicializálására:

- Azok a konstruktorok, amelyek nevükben tartalmazzák a „*Res*" szócskát, a megadott azonosítójú sztringerőforrásból (**resourcestring** deklaráció) inicializálják a *message* tulajdonságot.
- A nevükben „*Fmt*" szócskát tartalmazó konstruktorok paraméterként egy formátumsztringet fogadnak, ahol az *args* – típus nélküli nyitott tömb – paraméter tartalmazza a formázandó elemeket. Ekkor a *message* tulajdonság kezdeti értéke a ***SysUtils*** unitban található ***Format()*** függvény hívásával áll elő.

### 12.4.2 Kivételek kiváltása - a raise utasítás

A kivételek általában valamilyen programhibát jeleznek, azonban a **raise** utasítás segítségével magunk is kiválthatjuk őket:

```
raise ⟨kivételpéldány⟩;
```

A **raise** kulcsszó után osztálytípusú objektumnak kell állnia. Amikor fellép egy kivétel, a kivételkezelő átveszi a kivételobjektumot. Miután a kivételkezelés megtörtént, a kivételobjektum automatikusan, – a ***Destroy()*** destruktor hívását követően – törlődik.

Fontos megjegyeznünk, hogy a **raise** utasítás a paraméterként objektumot, nem pedig osztályt vár. Általában a kivételobjektumot közvetlenül az utasításban hozzuk létre, a megfelelő kivételosztály *Create()* konstruktorának hívásával. A paraméter nélküli **raise** utasítás újra kiváltja az aktuális kivételt.

A **raise** utasítástól a vezérlés sohase kerül vissza a hiba keletkezésének helyére. A **raise** a vezérlést annak a legbelső kivételkezelőnek adja, amely alkalmas az adott osztályú kivételek kezelésére.

A következő *Str2IntR()* függvény a karaktersorozatban tárolt számot egésszé alakítja. Amennyiben az eredmény kilép a megadott intervallumból, egy ***ERangeError*** kivétel jön létre:

```
resourcestring
   fmtstr = '%d nincs a megadott [%d..%d] intervallumban';
function Str2IntR(const s: string; min, max: integer): integer;
begin
   result := StrToInt(s);
   if (result < min) or (result > max) then
      raise ERangeError.CreateResFmt(@fmtstr, [result, min, max]);
end;
begin
    Str2Intr('123', 12, 23);
end.
```

A fenti függvényt használó program futtatásakor az alábbi üzenet informál a kezeletlen kivétel fellépéséről:

```
An unhandled exception occurred at $00401101 :
ERangeError : 123 nincs a megadott [12..23] intervallumban
```

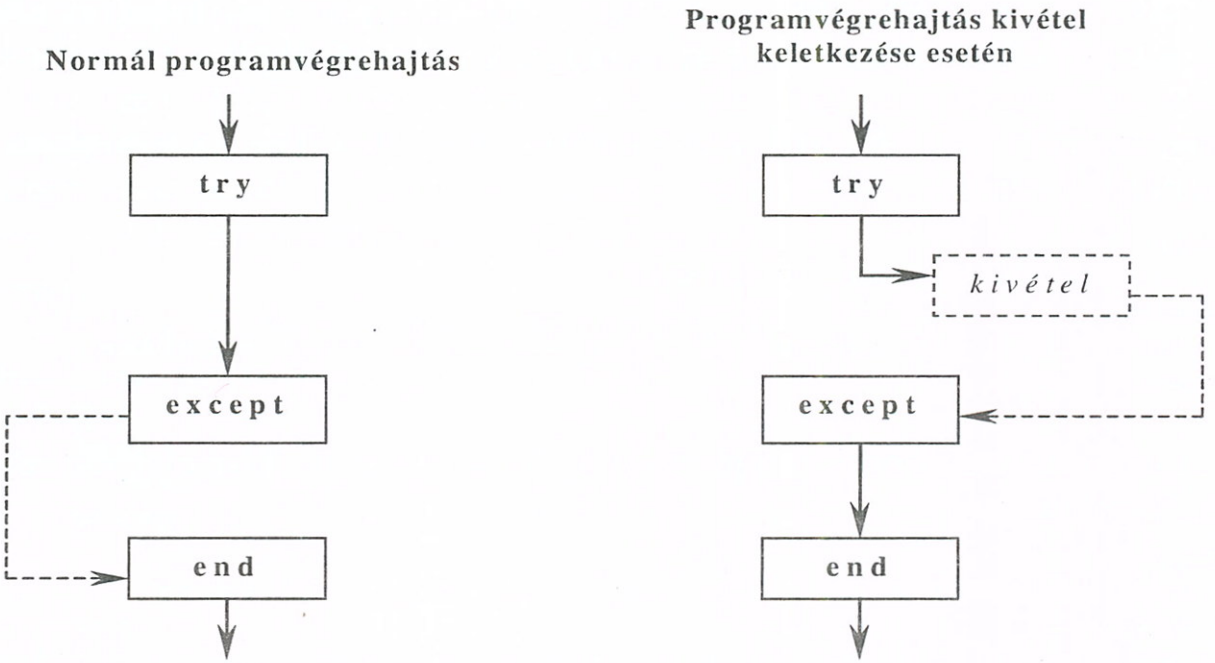
### 12.4.3 A try...except utasítás

A kivételek kezelését a **try**...**except** utasítás segítségével végezhetjük, melynek általános felépítése:

```
try
   utasítások
except
   kivételblokk
(else
   kivételblokk)
end;
```

A *kivételblokk* utasításokat és kivételkezelőket tartalmazhat. A kivételkezelőket a következő alakban adhatjuk meg:

```
on (azonosító:) osztálytípus do utasítás;
```




A futtató rendszer egymás után végrehajtja a **try...except** blokkban megadott utasításokat. Ha a végrehajtás során nem lép fel hiba, akkor a *kivételblokk* figyelmen kívül marad, és a vezérlés a **try...except** utasítást záró **end** után álló utasításra adódik.

Az utasítás **except...end** része kivételkezelőket tartalmaz. Valamely kivételkezelő futtatását csak a **try...except** részben, illetve a **try...except** részből hívott alprogramban fellépő hiba, vagy a **raise** utasítás végrehajtása eredményezi.

Amikor fellép egy kivétel, a vezérlés ahhoz a legbelső kivételkezelőhöz kerül, amely alkalmas az adott osztályú kivétel kezelésére. A megfelelő kivételkezelő keresése az utolsóként végrehajtott **try...except** utasításban kezdődik. Ha ez nem tartalmaz megfelelő kezelőt, akkor a keresés az egymásba ágyazott **try...except** utasításokon keresztül, kifelé haladva folytatódik. Amennyiben egyetlen aktív **try...except** utasítás sem tartalmazza a szükséges kezelőt, akkor a program futása hibaüzenettel megszakad.

A megfelelő kivételkezelő keresése a *kivételblokk* **on...do** kezelőinek vizsgálatával történik. Az **on...do** kivételkezelőket a beírás sorrendjében ellenőrzi a rendszer, kiválasztva az első olyan kezelőt, amely típuskompatíbilis az aktuális kivétellel. Ezért ajánlott, hogy a kezelőlistában először mindig a származtatott kivételek kezelőit adjuk meg:

```
try
  ...
except
  on EZeroDivide do NullavalOsztas;    // nullával való osztás
  on EOverflow do Tulcsordulas;        // túlcsordulás
  on EMathError do MatematikaiHiba;    // matematikai hiba
                                       // kivételek kezelése
end;
```

Ha a *kivételblokk* tartalmaz **else** részt is, akkor az **except** részben folytatott sikeretlen keresés után, az **else** részben megadott utasítások hajtódnak végre.

Amennyiben a *kivételblokk* csak utasításokat tartalmaz, akkor azok minden kivétel esetén végrehajtódnak:

```
try
  a:=StrToInt(szamszovegben);
except
  a:=0; //A try részben fellépő összes kivétel kezelése
end;
```

Az **on...do** kivételkezelőben szereplő opcionális *azonosító*t a kivételobjektum azonosítására használhatjuk:

```
try
  ...
except
  on Kivetel: Exception do writeln(Kivetel.Message);
end;
```

### *Kivételek ismételt előidézése*

Amikor egy alprogram, a benne fellépő kivételekhez nem tartalmaz kezelőt a **try...except** blokkban, a kivételeket továbbítanunk kell az alprogramot hívó külső kivételkezelő utasításnak, melynek **except** része tartalmazhat kezelőket az adott típusú kivételek feldolgozására. Ezt a megoldást az eredeti kivétel ismételt kiváltásának nevezzük. Az aktuális kivételt ismételten előidézhetjük, ha **try...except** utasítás **except** részében szerepeltetjük a paraméter nélküli **raise** utasítást.

Ha a kivételt újraaktiváló alprogram bármilyen erőforrást foglal, akkor a kivétel ismételt előidézése előtt fel kell szabadítanunk a lefoglalt erőforrásokat:

```
DLista := TLista.Create;  // a lista létrehozása
try
  ...
except
  DLista.Free;  // a lista felszabadítása
  raise;        // az eredeti kivétel ismételt kiváltása
end;
```

### *Beágyazott kivételek*

A kivételkezelőben végrehajtódó kód maga is kiválthat és kezelhet kivételeket. Ha a kivételkezelőben kiváltott kivételeket a kezelőn belül dolgozzuk fel, akkor ezek semmilyen hatással sincsenek az eredeti kivételre. Ha azonban a kivételkezelőben fellépő kivétel átterjed a kezelő határain kívülre, akkor az eredeti kivétel elvész. (Ezt a jelenséget a kivételek átváltására használhatjuk.)

Az alábbi példában szereplő *Kotangens()* függvényben az ***EDivByZero*** kivétel létrejötte esetén – ha a ***sin()*** függvény *0* értékű – a kivételkezelő egy *EKotangensError* kivételt hoz létre. Mivel ennek a kivételnek a kezelőjét nem tartalmazza függvény, ezért ez a kivétel az eredeti kivételkezelőn kívülre kerül. Az ***EDivByZero*** ennek következtében megszűnik, és a függvényt hívó programrész számára a *Kotangens()* függvény egy *EKotangensError* kivétellel tér vissza.

```
type
  EKotangensError = class(EMathError);
function Kotangens(x: real): real;
  begin
    try
      result := cos(x)/sin(x);
    except
        on EDivByZero do
        raise EKotangensError.Create('Hibás argumentum (tan)');
    end;
end;
```

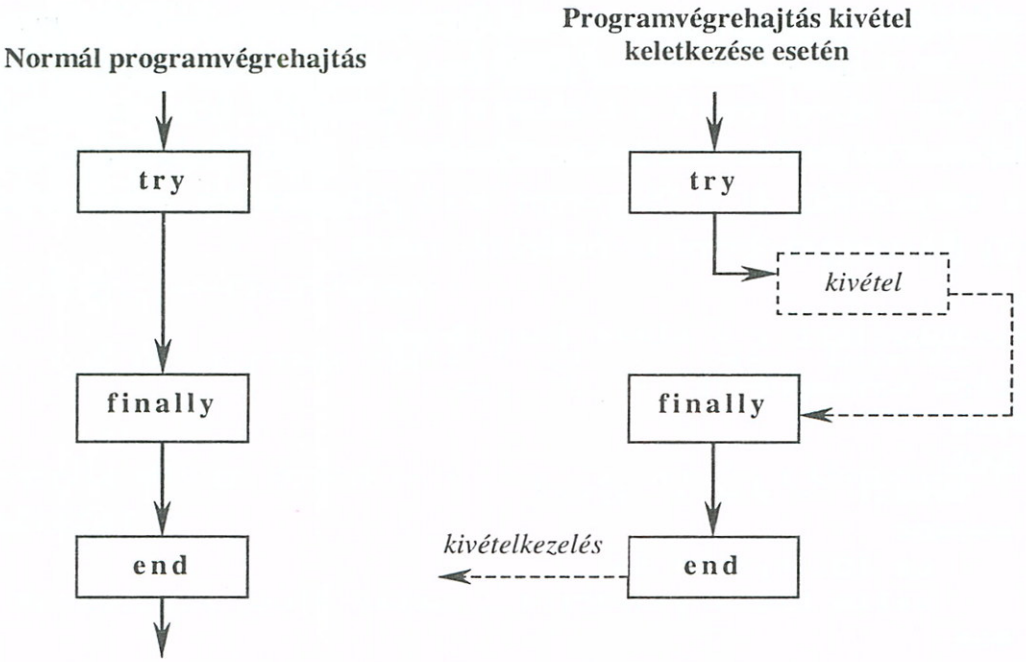
### 12.4.4 A try... finally utasítás

Amikor a programban valamilyen erőforrást használunk, szükséges lehet annak biztosítása, hogy – mind a normál, mind pedig kivétellel megszakított programvégrehajtás esetén – felszabadítsuk a lefoglalt erőforrást. Például, a fájl megnyitását és feldolgozását követően mindig gondoskodnunk kell az állomány lezárásáról, bárhogy is fejeződött be a feldolgozás. A fentiekhez hasonló esetekben használjuk a **try**...**finally** utasítást, melynek általános formája:

```
try
   utasítások
finally
   utasítások
end;
```

A futtató rendszer egymás után végrehajtja a **try** és a **finally** szavak között álló utasításokat. Abban az esetben, ha a végrehajtás folyamán nem lépett fel semmilyen kivétel, a program futása a **finally** és **end** között álló utasításokkal folytatódik.

Amennyiben az első rész végrehajtása során kivétel keletkezik, a vezérlés átadódik a **finally** részhez, és az ott megadott utasítások végrehajtása után a kivétel automatikusan újra fellép (melynek feldolgozása általában egy külső **try**...**except** utasításban történik). Ezzel a megoldással a **finally**...**end** rész utasításai mindig végrehajtódnak, függetlenül a **try**...**finally** rész feldolgozásának sikerességétől.




A következő példában kétféleképpen is gondoskodunk arról, hogy egy sikeresen megnyitott állományt minden esetben lezárjunk:

```
Filenyitas(F);
try
  Feldolgozas(F);
finally
  Lezaras(F);
end;
```

```
try
  Filenyitas(F);
  try
    Feldolgozas(F);
  finally
    Lezaras(F);
  end;
except
  Hibakezeles;
end;
```

### 12.4.5 Előre definiált kivételek

Az alábbi táblázatban összefoglaltuk a ***SysUtils*** modulban definiált kivételeket és a kivételeket kiváltó állapotokat. Ahol külön nincs megemlítve, a kivételek egyenes leszármazottai az ***Exception*** osztálynak.

| *Kivételosztály* | *Leírás* |
|---|---|
| ***EAbort*** | Az ***Abort()*** eljárás által kiváltott „csendes" kivétel. |
| ***EAbstractError*** | A kivétel jelzi, hogy kísérlet történt absztrakt metódus hívására. |
| ***EAccessViolation*** | Az ***EExternal*** leszármazottja. Akkor lép fel, ha az alkalmazás érvénytelen memóriacímet próbál elérni. |
| ***EControlC*** | Az ***EExternal*** kivételből származtatott kivétel jelzi, hogy a konzol alkalmazásunk futása során használták a *<Ctrl+C>* billentyűket. |
| ***EConvertError*** | Akkor lép fel, ha egy konverziós függvény (***StrToInt***, ***StrToFloat***, stb.) nem tudja végrehajtani a szükséges átalakítást. |

| *Kivételosztály* | *Leírás* *(folytatás)* |
|---|---|
| ***EDivByZero*** | Az ***EIntError*** leszármazottja. Egész számok nullával való osztásakor keletkezik. |
| ***EInOutError*** | Állományok I/O műveletei által okozott hibák esetén lép fel. Az ***EInOutError*** kivétel definiálja az I/O művelet hibakódját tartalmazó *ErrorCode* mezőt, melynek értéke megfelel az ***IOResult()*** szabványos függvény által visszaadott értékeknek. |
| ***EIntError*** | Az egész aritmetikai kivételek közös őse. |
| ***EIntfCastError*** | Hibás interface-konverziót jelző kivétel. |
| ***EIntOverflow*** | Az ***EIntError*** leszármazottja. Akkor lép fel, ha a túlcsordulás-ellenőrzés hibát jelez a {*$Q+*} állapotban fordított kódban. |
| ***EInvalidCast*** | Akkor lép fel, ha az **as** operátor bal oldalán megadott objektum osztálya nem egyezik meg az operátor jobb oldalán megadott osztály típusával. |
| ***EInvalidOp*** | Az ***EMathError*** leszármazottja. Akkor keletkezik, ha egy érvénytelen matematikai művelet hajtódik végre, mint például négyzetgyökvonás negatív számból. |
| ***EInvalidPointer*** | Akkor keletkezik, ha az alkalmazás megkísérel felszabadítani egy érvénytelen mutatóval kijelölt memóriaterületet. |
| ***EMathError*** | A lebegőpontos matematikai kivételek őse. |
| ***EOverflow*** | Az ***EMathError*** leszármazottja. Akkor keletkezik, ha egy lebegőpontos művelet túlcsorduláshoz vezet. |
| ***EPrivilege*** | A processzor védelmi rendszerének megsértését jelző kivétel. |
| ***ERangeError*** | Az ***EIntError*** leszármazottja. Akkor lép fel, amikor az intervallum-ellenőrzés hibát talál (a {*$R+*} állapotban fordított kódrészletben). |
| ***EStackOverflow*** | Akkor keletkezik, ha nincs elég hely a veremben (*stack*) az alprog-ramok lokális változóinak elhelyezésére. |
| ***EUnderflow*** | Az ***EMathError*** leszármazottja. Alapértelmezés szerint az alulcsordulást okozó lebegőpontos műveletek egyszerűen nullát eredményeznek. |
| ***EZeroDivide*** | Az ***EMathError*** leszármazottja. Nullával való osztás esetén keletkezik. |

A kivételkezelés összefoglalásaként készítsünk egy egyszerű számológép programot, amelynek hibás eseményeit a kivételkezelés segítségével dolgozzuk fel! (*KivetelKalk*)

Az egyszerű adatbevitel érdekében a műveletek az alábbi formában kell kijelölni:

```
:művjel operandus1 operandus2
```

```
{$mode ObjFPC}
program KivetelKalk;
uses  SysUtils, Math;
```

```
type
  integer = int64;

 // Saját kivételosztály létrehozása
 EMJelHiba = class(ERangeError);

const
  mjelek = ['+', '-', '*', '/', '^'];
  kjelek = ['q','x','Q','X'];

var
  a, b   :  integer;  // szám1, szám2
  c      :  integer;  // az eredmény
  op     :  char;     // az operátor
  vege   :  boolean;

// Adatbevitel
procedure Beolvas;
begin
    write(':');
    read(op);
    // Rossz operátor esetén saját kivétel kiváltása
    if not (op in (mjelek + kjelek)) then
     begin
       readln;
       raise EMJelHiba.Create('A műveleti jel csak +,-,*,/,^ lehet!');
     end;
    { Program befejezését jelző operátor esetében az EAbort kivételt
      kiváltó Abort függvény hívása }
    if op in kjelek then Abort;
    readln(a, b);
end;

// Számításokat végző függvény
function Szamol: Integer;
begin
  result:=0;
  case op of
    '+': result:=a+b;                    // összeadás
    '-': result:=a-b;                    // kivonás
    '*': result:=a*b;                    // szorzás
    '/': result:=a div b;                // osztás
    '^': result:=trunc(power(a,b));      // hatványozás
    end;
end;

// A számítási eredményt kiíró eljárás
procedure Kiir;
begin
  writeln(a,op,b,'=',c);
end;

// A főprogram
begin
  vege := False;
```

```
  repeat
      try
        Beolvas;
        c:=Szamol;
        Kiir;
      except
         // Ötféle kivétel kezelése
         on E: EMJelHiba do writeln(E.Message);
         on EDivByZero do writeln('Hiba: nullával osztás!');
         on EIntOverflow do writeln('Hiba: túlcsordulás!');
         on EInvalidOp do writeln('Matematikai hiba!');
         on EAbort do vege := True;
      else
        // Minden más kivétel esetén
         writeln('Hiba!');
         readln;
      end;
  until vege;
end.
```

*A program futásának eredménye:*

```
:+ 12 23
12+23=35
:/ 12 0
Hiba: nullával osztás!
:^ 123 123
Matematikai hiba!
:^ 12 2
12^2=144
:@ 34 56
A műveleti jel csak +,-,*,/,^ lehet!
:- 123 23
123-23=100
:* 12 23
12*23=276
:q
```

## 12.5 Szálak programozása

A többprogramos operációs rendszerek lehetővé teszik, hogy a programok (*process*, *task*) több párhuzamosan működő programszálban (*thread*) végezzék a tevékenységüket.

A Free Pascal rendszerben kétféleképpen is készíthetünk többszálú, „párhuzamos" feldolgozást biztosító alkalmazásokat, lehetővé téve ezzel a hatékonyabb fejlesztést. Először megnézzük a futásidejű könyvtár alprogramokkal támogatott megoldását, majd pedig áttekintjük a többszálú programozás objektum-orientált megközelítését.

### 12.5.1 Szálkezelés könyvtári hívásokkal

Ebben az esetben az egyes szálak tevékenységét

```
TThreadFunc = function(parameter : pointer) : longint;
```

típusú függvényekben kell megadnunk. A ***pointer*** típusú paraméter lehetővé teszi a szál indításához szükséges adatok átadását.

A szálak kezelését: indítását, felfüggesztését, megállítását stb. a ***System*** és a ***SysUtils*** unitok alprogramjai segítik. A szálak futását a ***BeginThread()*** függvény hívásával indíthatjuk el:

```
function BeginThread(ThreadFunction: TThreadFunc) : TThreadID;
function BeginThread(ThreadFunction: TThreadFunc;
                     p: pointer) : TThreadID;
function BeginThread(ThreadFunction: TThreadFunc; p: pointer;
                     var ThreadId: TThreadID) : TThreadID;
```

A függvény átdefiniált (*overloaded*) változatainak első paramétere a futtatandó függvény címe. A pointer típusú *p* paraméter értéke – illetve **nil** az első esetben – átadódik a szálfüggvénynek. A függvények a szálat azonosító egész értékkel térnek vissza, amely szükséges a további függvények használatához.

A programszál a megadott függvény végének elérésekor, illetve a függvényből hívott ***EndThread()*** eljárás hatására fejezi be futását.

```
procedure EndThread(ExitCode: DWord);
procedure EndThread;
```

A további szálműveleteket csak áttekintjük:

- Várakozás a megadott szál lefutásáig, de legfeljebb adott idő (*ms*) elteltéig:
  ```
  function WaitForThreadTerminate(threadHandle: TThreadID;
                                  TimeoutMs: LongInt):DWord;
  ```
- A megadott szál futásának megszakítása:
  ```
  function KillThread(threadHandle: TThreadID):DWord;
  ```
- A megadott szál futásának felfüggesztése:
  ```
  function SuspendThread(threadHandle: TThreadID):DWord;
  ```
- A felfüggesztett szál futásának folytatása:
  ```
  function ResumeThread(threadHandle: TThreadID):DWord;
  ```
- Az adott szál prioritásának lekérdezése:
  ```
  function ThreadGetPriority(threadHandle: TThreadID):LongInt;
  ```
- Az adott szál prioritásának (*-15..15*) beállítása:
  ```
  function ThreadSetPriority(threadHandle: TThreadID;
                             Prio: LongInt):Boolean;
  ```
- Az éppen futó szál azonosítójának lekérdezése:
  ```
  function GetCurrentThreadId: TThreadID;
  ```
- Egy futó szálból a vezérlést átengedhetjük egy másik szálnak:
  ```
  procedure ThreadSwitch;
  ```

A szálak elérik a program változóit, ez azonban nem minden esetben problémamentes. Természetesen a szálfüggvény lokális változói minden szálpéldány esetén létrejönnek. A program globális változóit azonban megosztva használják a szálak. Amennyiben a **var** deklaráció helyett, a **threadvar** után deklarálunk globális változókat, akkor azok másolatát minden induló szál megkapja, az alapértelmezés szerinti kezdőértékkel.

Ha a szálak normál globális változóval végeznek alapműveletek, akkor ajánlott használni a ***Sysutils*** modul *thread-safe* (szálbiztos) megoldásait:

- A ***Dec()*** eljárás megfelelője:
  ```
  function InterLockedDecrement(var Target: LongInt):LongInt;
  ```
- Az ***Inc()*** eljárás megfelelője:
  ```
  function InterLockedIncrement(var Target: LongInt):LongInt;
  ```
- A *Target* paraméternek megfelelteti a *Source* értékét:
  ```
  function InterLockedExchange(var Target: LongInt;
                               Source: LongInt):LongInt;
  ```
- A *Target* paraméterhez hozzáadja a *Source* értékét:
  ```
  function InterLockedExchangeAdd(var Target: LongInt;
                                  Source: LongInt):LongInt;
  ```

Készítsünk szálfüggvényt, amely adott intervallumon megjeleníti a prímszámokat! Futtassuk két szálban a keresést, és számoljuk meg szálanként és együtt a találatok számát! (*PrimSzalak1*)

```
{$mode ObjFPC}
program PrimSzalak1;
uses SysUtils {$ifdef unix},cthreads{$endif};

type params = record
          khatar, vhatar : longint;
          id : integer;
     end;
     pparams = ^params;

var
  osszdb: longint = 0;

threadvar
  darab : longint;

function Prim(p:longint):boolean;
var m, i : longint;
begin
    m:=round(sqrt(p));
    result:=true;
    for i:=2 to m do
      if p mod i = 0 then
        begin
          result:=false;
          break;
        end;
end;
```

```
function Primszamok(pth : pointer):longint;
var
 p : longint;
begin
 write('#start : ':13,pparams(pth)^.id:2,'#');
 darab:=0;
 p:=pparams(pth)^.khatar;
 while p<=pparams(pth)^.vhatar do
   begin
     if Prim(p) then
       begin
         write(p:8);
         inc(darab);
         InterLockedIncrement(osszdb);
       end;
     inc(p);
   end;
   write('#stop:':9,pparams(pth)^.id:2, darab:4,'#');
end;

var
  thid1, thid2 : TThreadID;
  par1, par2 : params;
begin
   par1.khatar := 100;  par1.vhatar := 500; par1.id:=1;
   thid1 := BeginThread(@Primszamok,pointer(@par1));
   par2.khatar := 10000;  par2.vhatar := 10650; par2.id:=2;
   thid2 := BeginThread(@Primszamok,pointer(@par2));
   WaitForThreadTerminate(thid1, 1000);
   WaitForThreadTerminate(thid2, 1000);
   writeln;
   writeln('A két szál összesen ',osszdb, ' prímszámot talált');
   readln;
end.
```

*A program futásának eredménye:*

```
    #start :  1#     101     103     107     109     113     127    #start :  2#
     131     137   10007     139   10009     149   10037     151   10039     157
   10061     163   10067     167   10069     173   10079     179   10091     181
   10093   10099     191   10103     193   10111     197   10133     199   10139
     211   10141     223   10151     227   10159     229   10163     233   10169
     239   10177     241   10181     251   10193     257   10211     263   10223
     269   10243     271   10247     277   10253     281   10259     283   10267
     293     307   10271     311   10273     313   10289     317   10301     331
   10303     337   10313     347   10321     349   10331     353   10333     359
   10337     367   10343     373   10357     379   10369     383   10391     389
   10399   10427     397   10429     401   10433     409   10453     419   10457
     421     431   10459     433   10463     439   10477     443   10487     449
   10499     457   10501     461   10513     463   10529     467   10531     479
   10559     487   10567     491     499   10589   #stop: 1  70#   10597   10601
   10607   10613   10627   10631   10639   #stop: 2  69#
A két szál összesen 139 prímszámot talált
```

A Free Pascal megteremti annak lehetőségét, hogy azonos erőforrást használó szálak közül mindig csak egy férjen hozzá az erőforráshoz. Ehhez négy speciális hívással ún.

*kritikus programrészt* alakíthatunk ki a mellékelt példa szerint. A programstruktúra biztosítja, hogy a *kritikus programrészt* mindig csak egy szál hajtja végre.

```
{$mode ObjFPC}
var
  kritikus : TRTLCriticalSection;

procedure KritikusEljaras;
begin
  EnterCriticalSection(kritikus);
  try
    // kritikus programrész
  finally
    LeaveCriticalSection(kritikus);
  end;
end;

function SzalFuggveny(p:pointer):longint;
begin
  // ...
  KritikusEljaras;
  // ...
end;

var
  thid1, thid2 : TThreadID;
begin
  InitCriticalSection(kritikus);
    // szálak indítása
    thid1 := BeginThread(@SzalFuggveny);
    thid2 := BeginThread(@SzalFuggveny);
    WaitForThreadTerminate(thid1, 1000);
    WaitForThreadTerminate(thid2, 1000);
  DoneCriticalSection(kritikus);
  readln;
end.
```

### 12.5.2 A TThread osztály felhasználása

A szálak kezeléséhez szükséges műveleteket és tulajdonságokat a ***Classes*** modul ***TThread*** osztálya fogja csokorba, melynek nyilvános része:

```
type
TThread=class
 public
   constructor Create();
   destructor Destroy;override;
   // szálműveletek
   procedure Resume;        // a felfüggesztett szál továbbindítása
   procedure Suspend;       // a szál futásának felfüggesztése
   procedure Terminate;     // a szál futásának megszakítása
   function WaitFor;        // várakozás a szál lefutásáig
```

```
    // tulajdonságok, r/w - írható/olvasható
    // r/w a szál a lefutása után szabadítsa fel önmagát
    property FreeOnTerminate:Boolean;
    property ThreadID:TThreadID;          // r szálazonosító
    property Priority:TThreadPriority;    // r/w a szál prioritása
    property Suspended:Boolean;           // r/w felfüggesztett állapot
    // r/w a szál lefutásakor aktiválódó esemény
    property OnTerminate:TNotifyEvent;
    // r a futás során fellépő kivétel
    property FatalException:TObject;
end;
```

A szálak prioritását a ***TThreadPriority*** típus értékei szabályozzák:

| *azonosító* | *prioritás* |
|---|---|
| *tpIdle* | „üresjárati", |
| *tpLowest* | a legalacsonyabb, |
| *tpLower* | alacsony, |
| *tpNormal* | normál, |
| *tpHigher* | magas, |
| *tpHighest* | a legmagasabb, |
| *tpTimeCritical* | valósidejű. |

A *tpIdle* azt jelzi, hogy a szál csak akkor fut, ha minden más folyamat várakozó állapotban van, vagyis semmilyen más szálat nem szakít meg.

A szálak objektum-orientált kezeléséhez a ***TThread*** osztályból saját szálosztályt kell származtatnunk, és újra kell definiálnunk az öröklött, **protected** elérésű ***Execute()*** metódust. Az ***Execute()*** metódusban kell elhelyeznünk azt a programkódot, amelyet önálló szálban kívánunk futtatni.

```
type
  TSzal = class(TThread)
  // ...
  protected
    procedure Execute; override;
  end;
```

Készítsünk el a *PrimSzalak1* program szálobjektumokat használó változatát! (*PrimSzalak2*)

```
{$mode objfpc}
program PrimSzalak2;

uses
  Classes, SysUtils;

var
  osszdb : longint;
```

```
type
  TPrimSzal = class(TThread)
  private
    khatar, vhatar, darab, id : longint;
  protected
    procedure Execute; override;
    procedure Primszamok;
  public
    constructor Create(kezdet, veg, azonosito :longint ;
                       felfuggesztett : Boolean = False);
    function Prim(p:longint):boolean;
  end;

constructor TPrimSzal.Create(kezdet, veg , azonosito : longint ;
                             felfuggesztett : Boolean=False);
begin
  khatar := kezdet;  vhatar := veg;
  id := azonosito;   darab:=0;
  inherited Create(felfuggesztett);
end;

procedure TPrimSzal.Execute;
begin
     Primszamok;
end;

function TPrimSzal.Prim(p:longint):boolean;
var m, i : longint;
begin
    m:=round(sqrt(p));
    result:=true;
    for i:=2 to m do
      if p mod i = 0 then
        begin
          result:=false;
          break;
        end;
end;

procedure TPrimSzal.Primszamok;
var p : longint;
begin
 write('#start : ':13,id:2,'#');
 p:=khatar;
 while p<=vhatar do
   begin
     if Prim(p) then
       begin
         write(p:8);  inc(darab);
         InterLockedIncrement(osszdb);
       end;
    inc(p);
   end;
   write('#stop:':9,id:2, darab:4,'#');
end;
```

```
var
  PrimSzal1, PrimSzal2 : TPrimSzal;
begin
   PrimSzal1 := TPrimSzal.Create(100, 500, 1);
   PrimSzal1.priority:=tpnormal;
   PrimSzal2 := TPrimSzal.Create(10000,10650,2);
   PrimSzal2.priority:=tpnormal;
   PrimSzal1.WaitFor;
   PrimSzal2.WaitFor;
   writeln;
   writeln('A két szál összesen ',osszdb, ' prímszámot talált');
   readln;
end.
```

*A program futásának eredménye:*

```
   #start :  1#      101      103    #start :  2#      107    10007      109    10009
    113    10037      127    10039      131    10061      137    10067      139    10069
    149    10079      151    10091      157    10093      163    10099      167    10103
    173      179    10111      181    10133      191    10139      193    10141      197
  10151      199    10159      211    10163      223    10169      227    10177      229
  10181      233    10193      239      241    10211      251    10223      257    10243
    263    10247      269    10253      271    10259      277    10267      281    10271
    283    10273      293    10289      307    10301      311    10303      313    10313
    317      331    10321      337    10331      347    10333      349    10337      353
  10343      359    10357      367    10369      373    10391      379    10399      383
  10427      389    10429      397    10433      401    10453      409    10457      419
  10459    10463      421    10477      431    10487      433    10499      439    10501
    443    10513      449    10529      457    10531      461    10559      463    10567
    467      479    10589      487    10597      491      499  #stop: 1  70#    10601
  10607    10613    10627    10631    10639  #stop: 2  69#
A két szál összesen 139 prímszámot talált
```

## 12.6 A Classes modul néhány hasznos osztálya

A *Classes* unitban egy sor, jól használható osztályt találunk, melyek nagyban segítik a vizuális programozási felületek Free Pascalra alapozott fejlesztését. Az előző rész *TThread* osztályának megismerése után, csupán néhány további osztályra szeretnénk felhívni a figyelmet, példák segítségével.

| *Osztály* | *Ősosztály* | *Rövid leírás* |
|---|---|---|
| *TList* | *TObject* | mutatók listáját kezelő osztály, |
| *TStream* | *TObject* | az adatfolyam-osztályok (*TFileStream*, *TMemoryStream*, *TResourceStream*, *TStringStream*) alaposztálya, |
| *TStrings* | *TPersistent* | a sztringek tárolását segítő absztrakt alaposztály, |
| *TStringList* | *TStrings* | sztringek listáját kezelő osztály, |
| *TCollection* | *TPersistent* | a névvel ellátott objektumok gyűjteményét kezelő osztály, |
| *TBits* | *TObject* | bitek indexelt tárolását és kezelését segítő osztály. |

A *TList* osztály felhasználásával készítsünk programot, egészre mutató pointerek listájának kezelésére! Végezzünk műveleteket a listával és jelenítsük meg a tárolt adatokat! (*EgeszLista*)

A megoldás során joggal vetődik fel a kérdés, miért használunk egyetlen egész szám tárolására legalább kétszer akkora memóriát, mint amennyi az egész változók bájtmérete. A példában a listakezelés a lényeg, az egészek helyett pedig tetszőleges méretű adatstruktúrákat is hasonlóan kezelhetnénk. A megoldásban a lista létrehozásán és törlésén túlmenően az alábbi műveleteket valósítottuk meg:

| *Listaművelet* | *Megoldás* |
|---|---|
| Elem hozzáadása a lista végére | a ***TList Add()*** metódusa, |
| Adott indexű elem törlése | a ***TList Delete()*** metódusa, |
| Beszúrás adott indexű helyre | a ***TList Insert()*** metódusa, |
| Az elemek rendezése | a ***TList Sort()*** metódusa, saját *Osszehasonlit()* függvénnyel, |
| Érték keresése a listában | saját *Keres()* függvény, |
| A listaelemek megjelenítése | saját *Megjelenit()* eljárás. |

```
{$mode ObjFPC}
program EgeszLista;
uses Classes;

type
  PInteger = ^integer;

procedure Megjelenit(Lista : TList);
var i : integer;
begin
  for i:=0 to Lista.Count-1 do // indexek: 0..(count-1)
    write(PInteger(Lista.Items[i])^:4);
  writeln;
end;

// A függvény igaz értékkel tér vissza,
// ha a megadott érték már szerepel a listában
function Keres(ertek : Integer; Lista : TList) : Boolean;
var i : integer;
begin
  for i:=0 to Lista.Count-1 do
    if PInteger(Lista.Items[i])^=ertek then
      exit(true);
  exit(false);
end;

// A rendezéshez használt összehasonlító függvény, melynek értéke
// negatív, ha p1^<p2^, nulla, ha p1^=p2^ és pozitív, ha p1^>p2^
function Osszehasonlit(p1, p2: pointer): integer;
begin
  result := PInteger(p1)^ - PInteger(p2)^;
end;

// A főprogram
var
  SzLista : TList;
  PSzam   : PInteger;  // mutató az egész számra
  szam, i : integer;
```

```
begin
  randomize;
  SzLista := TList.Create;  // a listaobjektum létrehozása
  try
    // a dinamikusan létrehozott egész számokat
    // elhelyezzük a listában, ismétlődés nélkül
    for i:= 1 to Random(15)+10 do
      begin
        szam := random(100)+100;          // 100..199
        if not keres(szam,SzLista) then  // nincs benne
           begin
             new(PSzam);                  // az egész szám létrehozása
             PSzam^:=szam;
             SzLista.Add(PSzam);          // hozzáadása a listához
          end;
      end;
    Megjelenit(SzLista);
    // az elemek rendezése a mutatók mozgatásával
    SzLista.Sort(@Osszehasonlit);
    Megjelenit(SzLista);
    // minden második elemet törlünk a listából
    i:=1;
    while i<SzLista.Count do
      begin
        SzLista.Delete(i);
        inc(i);
      end;
    Megjelenit(SzLista);
    // beszúrunk 100-at az első elem elé.
    // 199-et az utolsó elem után
    new(PSzam); PSzam^ := 100;
    SzLista.Insert(0, PSzam);
    new(PSzam); PSzam^ := 199;
    SzLista.Insert(SzLista.Count, PSzam);
    Megjelenit(SzLista);
    // a listaelemek és a listaobjektum törlése
    for i:=0 to SzLista.count-1 do
      begin
        PSzam := SzLista.Items[i];
        dispose(PSzam);
     end;
  finally
    SzLista.Free;
  end;
  readln;
end.
```

*A program futásának eredménye:*

```
155 192 185 120 112 149 132 110 160
110 112 120 132 149 155 160 185 192
110 120 149 160 192
100 110 120 149 160 192 199
```

Írjunk kisbetűs, angol szavak ellenőrzését végző programot! A program töltse fel az *AngolSzavak.txt* állomány tartalmát egy rendezett ***TStringList*** típusú objektumba, és végezzen listaműveleteket! Jelenítsük meg a „*swe*" kezdetű szavakat, és határozzuk meg a hárombetűs szavak számát! (*StringLista*)

```
{$mode ObjFPC}
program StringLista;
uses Classes;
var
  szotar : TStringList;
  szo    : string;
  idx, db : Integer;
begin
  szotar := TStringList.Create;
  try
    szotar.Sorted := true;           // rendezett sztringlista
    szotar.Duplicates := dupignore; // nem engedjük az ismétlődést
    // A szótár feltöltése állományból
    szotar.LoadFromFile('AngolSzavak.txt');
    // Hozzáadunk két új szót
    szotar.Add ('inheritance');
    szotar.Add ('interface');
    writeln('A szótár ', szotar.Count, ' szót tartalmaz.');
    // Indul a szóellenőrzés
    repeat
       write('Kérek egy angol szót [kilépés: x]: '); readln(szo);
       szo:=lowercase(szo);          // kisbetűssé alakítjuk
       if szo = 'x' then break;
       idx := szotar.IndexOf (szo); // benne van-e a szótárban
       if idx >= 0 then
          writeln('benne van')
       else
         writeln('nics benne');
   until false;
   // Megjelenítjük a "swe" kezdetű szavakat, és
   // megszámoljuk, hogy hány hárombetűs szót tartalmaz a szótár
   db:=0;
   writeln('Az swe kezdetű szavak:');
   for idx:=0 to szotar.Count-1 do
    begin
       szo:=szotar[idx];
       if copy(szo,1,3)='swe' then write(szo:16);
       if length(szo)=3 then inc(db);
    end;
   writeln;
   writeln('A hárombetűs szavak száma: ',db:5);
  finally
    szotar.Free;
  end;
  readln;
end.
```

*A program futásának eredménye:*

```
A szótár 4269 szót tartalmaz.
Kérek egy angol szót [kilépés: x]: Class
benne van
Kérek egy angol szót [kilépés: x]: sweet
benne van
Kérek egy angol szót [kilépés: x]: nata
nics benne
Kérek egy angol szót [kilépés: x]: x
Az swe kezdetű szavak:
     swear       sweat      sweater        swede         sweden
   swedish       sweep        sweet       sweets
A hárombetűs szavak száma:   193
```

Készítsünk programot, amely típusos állományban tárolt valós adatokat különböző módon (fájlként, fájl- és memória-adatfolyamként) átlagol! (*Adatfolyamok*)

A megoldáshoz szükséges fájlkezelési ismereteket az elektronikus függelék tartalmazza. A példában a hangsúlyt az adatfolyamok (*stream*) kezelésére helyeztük. A memória-adatfolyamba az átlagolás előtt felmásoljuk az adatokat az állományból, de ezzel együtt is ez a leggyorsabb megoldás. Az időmérő eljárásban a ***SysUtils*** és a ***DateUtils*** modulok elemeit használjuk. Az adatokat ezres egységekben kérjük be, hogy ne kelljen túl nagy számokat megadni.

Az adatfolyamok mindig bájtfolyamként kezelik a perifériákat. Ezért az írás (***Write()***) és olvasás (***Read()***) műveletek során egy változót és egy méretet kell megadnunk. A művelet kiindulási bájtpozícióját a ***Seek()*** metódus hívásával állíthatjuk be.

```
{$mode ObjFPC}
program Adatfolyamok;
uses Classes, SysUtils, DateUtils;

type TSS=(start, stop);

procedure Idomero(ss: TSS; ido: TDateTime);
const
    starttime:Tdatetime=0;
var dt:integer;
begin
    case ss of
      start:  begin
                starttime:=ido;
                write('Start...  ');
              end;
      stop:   begin
                dt:=MilliSecondsBetween(ido,starttime);
                write('Stop ', dt/1000:7:4,' mp - ' );
              end;
    end;
end;
```

```
procedure Letrehoz(dbadat : integer);
var f : file of real;
    adat  : real;
    i, db : integer;
begin
    Randomize; // véletlen számokkal töltjük fel a fájlt
    idomero(start, time);
    assignfile(f,'Adatok.dat');
    rewrite(f);
    db:=1000 * dbadat div sizeof(real);   // az adatok mennyisége
    for i:=1 to db do
       begin
         adat:=(100000.0+random(maxint))/random(maxint);
         write(f,adat);
       end;
    closefile(f);
    idomero(stop, time);
    writeln;
end;

procedure OlvasFaljkent;
var f : file of real;
    db : integer = 0;
    sum, adat : real;
begin
    idomero(start, time);
    assignfile(f,'Adatok.dat');
    filemode:=0; // olvasás
    reset(f);
    sum:=0;
    while not eof(f) do
       begin
         read(f,adat);
         inc (db);
         sum:=sum+adat;
       end;
    closefile(f);
    idomero(stop, time);
    writeln('átlag: ', sum/db:10:6);
end;

procedure OlvasFajlAdatfolyamkent;
var fs : TFileStream;
    i, db : integer;
    sum, adat : real;
begin
    idomero(start, time);
    fs:=TFileSTream.create('Adatok.dat', fmOpenread);
    sum:=0;
    db:=fs.Size div sizeof(real);
    for i:=1 to db do
      begin
        fs.read(adat,sizeof(real));
        sum:=sum+adat;
      end;
```

```
    fs.Free;
    idomero(stop, time);
    writeln('átlag: ', sum/db:10:6);
end;

procedure OlvasMemoriaAdafolyamkent;
var
    ms : TMemoryStream;
    fs : TFileStream;
    p  : ^ real;
    i, db : integer;
    sum   : real = 0;
begin
    idomero(start, time);
    ms:=TMemoryStream.create;
    fs:=TFileSTream.create('Adatok.dat', fmOpenread);
    ms.SetSize(fs.Size);      // az adatokat a fájlból a memóriába
    ms.CopyFrom(fs,fs.Size); // másoljuk
    p:=ms.Memory;
    db:=fs.Size div sizeof(real);
    for i:=1 to db do
      begin
        sum:=sum+p^;
        inc(p);
      end;
    fs.Free;
    ms.free;
    idomero(stop, time);
    writeln('átlag: ', sum/db:10:6);
end;

var
    darab : integer;
begin
    repeat
       write('Kérem az ezeradatok mennyiségét : ');
       readln(darab);
    until darab>0;
    write('Létrehoz, File of real  '); Letrehoz(darab);
    write('Felolvas, File of real  '); OlvasFaljkent;
    write('Felolvas, TFileStream   '); OlvasFajlAdatfolyamkent;
    write('Felolvas, TMemoryStream '); OlvasMemoriaAdafolyamkent;
    readln;
end.
```

*A program futásának eredménye:*

```
Kérem az ezeradatok mennyiségét : 12230
Létrehoz, File of real  Start...  Stop  5.5940 mp -
Felolvas, File of real  Start...  Stop 13.0310 mp - átlag:   7.880968
Felolvas, TFileStream   Start...  Stop  3.2330 mp - átlag:   7.880968
Felolvas, TMemoryStream Start...  Stop  0.0930 mp - átlag:   7.880968
```

Tervezzünk tetszőleges elemszámú, egész számok tárolására alkalmas halmazosztályt, a ***TBits*** osztály felhasználásával! Külső függvényekkel valósítsuk meg az unió, a metszet és a különbség halmazműveleteket! (*EgeszHalmaz*)

A megoldásban a TBits típusú adatmező a halmaz elemeiről tárol 1-1 bitnyi információt. Például, *1000* elemű (*0..999*) halmaz esetén, az eleminformációk *125* bájtot foglalnak. A halmazműveletek külső függvényekben valósítottuk meg. Minden művelet eredménye új halmazban jelenik meg, amelyet nekünk kell megszüntetni.

```
{$mode ObjFPC}
program EgeszHalmaz;
uses Classes, Math;

// A THalmaz osztály definíciója
type
  THalmaz = class
     private
        hbit : TBits;
     public
       constructor Create(darab:integer);
       destructor Destroy; override;
       procedure SetElem(const x : array of integer); // értékadás
       procedure SetElem(const halmaz : THalmaz);     // értékadás
       // elem(ek) hozzáadása
       procedure Include(const x : array of integer);
       // elem(ek) kivétele
       procedure Exclude(const x : array of integer);
       procedure Show(s:string);
  end;

constructor THalmaz.Create(darab:integer);
begin
  hbit := TBits.Create(darab);  // minden bit 0 lesz
end;

destructor THalmaz.Destroy;
begin
  hbit.free;
end;

procedure THalmaz.SetElem(const x : array of integer);
var
  i : integer;
begin
  hbit.ClearAll; // minden bit törlése
  for i:=0 to high(x) do
    if x[i]<hbit.Size then hbit.SetOn(x[i]);
end;
```

```
procedure THalmaz.SetElem(const halmaz : THalmaz);
var
  i : integer;
begin
  hbit.ClearAll;
  for i:=0 to Min(hbit.size,halmaz.hbit.Size)-1 do
    hbit.bits[i]:=halmaz.hbit.bits[i];
end;

procedure THalmaz.Include(const x : array of integer);
var
  i : integer;
begin
  for i:=0 to high(x) do
    if x[i]<hbit.Size then hbit.SetOn(x[i]); // beállít
end;

procedure THalmaz.Exclude(const x : array of integer);
var
  i : integer;
begin
  for i:=0 to high(x) do
    if x[i]<hbit.Size then hbit.Clear(x[i]); // töröl
end;

procedure THalmaz.Show(s:string);
var
  i : integer;
begin
  write(s,'[');
  for i:=0 to hbit.size-1 do
    if hbit.bits[i] then write(i:5);
  writeln(' ]');
end;

// Nem metódus halmazműveletek
function Unio(h1, h2 : THalmaz) : THalmaz;
var
  i : integer;
begin
  result:=THalmaz.Create(Max(h1.hbit.size,h2.hbit.Size));
  result.SetElem(h1);
  for i:=0 to h2.hbit.Size-1 do
    if h2.hbit.bits[i] then result.hbit.bits[i]:=true;
end;

function Metszet(h1, h2 : THalmaz) : THalmaz;
var
  i : integer;
begin
  result:=THalmaz.Create(Min(h1.hbit.size,h2.hbit.Size));
  for i:=0 to result.hbit.Size-1  do
     result.hbit.bits[i]:= h1.hbit.bits[i] and h2.hbit.bits[i];
end;
```

```
function Kulonbseg(h1, h2 : THalmaz) : THalmaz;
var
  i : integer;
begin
  result:=THalmaz.Create(h1.hbit.size);
  result.SetElem(h1);
  for i:=0 to Min(h1.hbit.size,h2.hbit.Size)  do
     result.hbit.bits[i]:= h1.hbit.bits[i] and (not h2.hbit.bits[i]);
end;

var
  h1, h2, h3 : THalmaz;
begin
  h1 := THalmaz.Create(1000);       // 0..999
  h1.SetElem([35,12,7,23,123,678]);
  h2 := THalmaz.Create(3000);       // 0..2999
  h2.SetElem(h1);
  h1.Exclude([35]);
  h1.Show('h1 = ');
  h2.Include([1979, 2003, 2006]);
  h2.Show('h2 = ');

  h3:=Unio(h1,h2);
  h3.Show('h1 + h2 = ');
  h3.Free;

  h3:=Metszet(h1,h2);
  h3.Show('h1 + h2 = ');
  h3.Free;

  h3:=Kulonbseg(h2,h1);
  h3.Show('h2 - h1 = ');
  h3.Free;

  h3:=Kulonbseg(h1,h2);
  h3.Show('h1 - h2 = ');
  h3.Free;

  h1.Free;
  h2.Free;
  readln;
end.
```

*A program futásának eredménye:*

```
h1 = [    7   12   23  123  678 ]
h2 = [    7   12   23   35  123  678 1979 2003 2006 ]
h1 + h2 = [    7   12   23   35  123  678 1979 2003 2006 ]
h1 + h2 = [    7   12   23  123  678 ]
h2 - h1 = [   35 1979 2003 2006 ]
h1 - h2 = [ ]
```

# Elektronikus függelék a CD-mellékleten

Az elektronikus függelék PDF-formátumú állományai a CD-melléklet *\Konyv* mappájában találhatók. Olvasásukhoz az ***Adobe Reader*** programot kell telepíteni a számítógépre (*\AdobeReader* könyvtár vagy *www.adobereader.com*). A *100* könyvoldalnyi terjedelemben néhány kiegészítő fejezettel segítjük a Pascal nyelv jobb elsajátítását, alkalmazását, valamint a ***Free Pascal*** fejlesztői környezetével való önálló ismerkedést.

## F1. Karaktersorozat-kezelő függvények és eljárások

A fejezetben áttekintjük a Free Pascal ***System*** moduljában található, karaktersorozatok (sztringek) használatát segítő, legfontosabb függvényeket és eljárásokat. Eközben nem teszünk különbséget a hosszú (***AnsiString***) és a rövid sztringek (***ShortString***) között.

## F2. Adatállományok kezelése

Az adataink tárolásának egyik jól bevált módszere a fájlok (adatállományok) használata. A Pascal nyelv szempontjából az állományokat – a tartalmuk alapján – szöveges (*text*), típusos és típus nélküli fájlok csoportjára osztjuk. A fejezetben mindhárom fájltípussal részletekbe menően foglalkozunk, bemutatva a kezelésük lépéseit:

- az állomány megnyitása,
- a fájl tartalmának feldolgozása fájlműveletek (olvasás, írás, pozicionálás) felhasználásával.
- az állomány lezárása,

valamint az ehhez szükséges Pascal típusokat, eljárásokat és függvényeket.

## F3. Numerikus módszerek programozása

A programozás tanulásának és alkalmazásának hagyományosan legfontosabb területe a matematika. A matematikai feladatok végeláthatatlan sokaságából néhány numerikus módszer bemutatását, és Pascal nyelven történő megvalósítását, tűztük ki célul az F3. függelékben.

***Lineáris egyenletrendszerek*** megoldása témakörben foglalkozunk a *Gauss*-féle kiküszöbölési eljárással, azon belül a részleges és a teljes főelemkiválasztás módszerével. Bemutatjuk a lineáris egyenletrendszer LU dekompozícióval való megoldását, amely jól alkalmazható mátrix invertálásra és determináns kiszámításra is. Az iteratív eljárások közül bemutatjuk a *Gauss-Seidel* és a *Jacobi*-módszert.

***Egyismeretlenes nemlineáris egyenlet*** megoldási módszerei közül az alábbiakkal foglalkozunk:

- behatárolás intervallum-felezéssel,
- érintő (*Newton-Raphson*) módszer,
- húrmódszer,
- *Newton-Raphson* és a húr módszer együttes alkalmazása,
- szelőmódszer,
- a fokozatos közelítés (szukcesszív approximáció) módszere.

A ***függvények közelítésére*** szolgáló interpolációs módszereket és a kétparaméteres regressziót is tárgyaljuk. Az interpolációs algoritmusok közül megismerkedünk a lineáris interpoláció, a *Lagrange*-interpoláció és az *Aitken*-interpoláció módszerekkel. A kétparaméteres regresszió kiszámítását tízféle függvénytípusra ismertetjük.

A fejezet negyedik témaköre a ***numerikus integrálás***, az ún. numerikus kvadratúra. Ezen belül foglalkozunk:

- a *Newton-Cotes* kvadratúra módszerekkel: a téglalap formula, az egyszerű és az összetett trapéz formula, az érintő formula és az egyszerű és az összetett *Simpson*-formula,
- a *Romberg*-eljárással,
- a *Gauss*-típusú kvadratúrával.

***Függvények szélsőértékének*** megkeresése témakörében bemutatjuk az egyváltozós függvény intervallumba zárt minimumának megkeresését, az intervallum módszeres szűkítésével.

## F4. A Free Pascal IDE használata

Az F4. függelékben azokat az ismereteket tárjuk az *Olvasó* elé, amelyek elengedhetetlenek ahhoz, hogy a ***Free Pascal*** fejlesztői környezetében ismerkedjen meg a Pascal nyelvvel. Bemutatjuk a programok megírásához, futtatásához, mentéséhez és betöltéséhez szükséges lépéseket.

# Irodalomjegyzék


Kathleen Jensen – Niklaus Wirth:
**A PASCAL programozási nyelv**
*Műszaki Könyvkiadó, Budapest, 1988*

Pongor György:
**Szabványos PASCAL, Programozás és algoritmusok**
*Novotrade Rt., Budapest, 1988*

Benkő Tiborné – Benkő László – Tóth Bertalan – Varga Balázs:
**Programozzunk Turbo Pascal nyelven!**
*ComputerBooks Kft., Budapest, megjelenés: 1996, utolsó kiadás: 2004*

Ray Lischner:
**Delphi kézikönyv**
*O'reilly Inc., Kossuth Kiadó, Budapest, 2000*

Marco Cantù:
**Delphi 7 Mesteri szinten**
*Sybex Inc., Kiskapu Kft.,, Budapest, 2003*

Kuzmina Jekatyerina - Dr. Tamás Péter - Tóth Bertalan:
**Programozzunk Delphi 7 rendszerben!**
*ComputerBooks Kft., Budapest, 2003*

Michaël Van Canneyt:
**Programmers' manual for Free Pascal, version 2.0.4**
**Reference guide for Free Pascal, version 2.0.4**
**Run-Time Library (RTL) Reference guide**
**Free Component Library (FCL) Reference guide**
*2006. augusztus*

Michaël Van Canneyt - Florian Klämpfl:
**Users' manual for Free Pascal, version 2.0.4**
*2006. augusztus*

# Tárgymutató

## *A,Á*


abs(), 42
absolute, 171
abstract, 213
absztrakt metódusok, 213
adatbevitel, 27
adatrejtés, 202
Addr(), 170
aktuális érték, 15
AllocMem(), 168
állomány, 101
alprogramok, 101
ancestor, 205
and, 47
ANSI, 11
AnsiChar, 18
AnsiString, 22
aritmetikai műveletek, 40
as, 230
ASCII, 11
Assigned(), 170
azonosítók, 12


## *B*


begin, 8, 59
BeginThread(), 244
biteltoló műveletek, 50
bitenkénti logikai műveletek, 48
bitműveletek, 48
Boole-algebra, 95
boolean, 18, 120
break, 76


## *C*


carrige return, 27
case, 59
char, 18
ciklus utasítások
    for, 68
    repeat-until, 74
    while, 72
cím szerinti paraméterátadás, 104
címke, 58
class, 218, 224
class methods, 219
Classes unit, 247, 250
const, 25
constructor, 199
continue, 76
Copy(), 172
Create(), 219, 222
currency, 17


## *D*


data hiding, 202
DateUtils unit, 254
dec, 20
deklaráció, 7, 15
Delphi, 11, 220
descendant, 205
Destroy(), 219
destructor, 199
dinamikus élettartam, 115
dinamikus tömb, 171
Dispose(), 166, 166
div, 40
DLL, 160
double, 17


## *E,É*


EDivByZero, 239
egész adatok, 16
egységbezárás, 193
egyszerű típusok, 16
egyszerű utasítások, 57
EInOutError, 235
eljárás, 103
eljáráshívás, 110
eljárás-paraméter, 141

eljárástípus, 16
előző, 19
else, 238
elsőbbségi szabály, 39
EMathError, 234
end, 8, 12, 59
ERangeError, 236
ErrorCode, 235
érték szerinti paraméterátadás, 103
értékadás, 58
except...end, 237
Exception, 234, 235, 240
exception handler, 234
Execute(), 248
exports, 159, 160
external, 160

## *F*

false, 18
felsorolt típus, 21
feltételes utasítások
  case, 65
  if, 60
field, 101
file, 101
foglalt szavak, 11
for, 59
Format(), 235
forward, 116
főprogram, 7
Free Pascal, 1, 3
Free(), 219
FreeMem(), 168
function, 103
futásidejű könyvtár, 234
függvény, 103
függvény azonosító, 105
függvény paraméterei, 105
függvény törzse, 105
függvényhívás, 106
függvény-paraméter, 141

## *G*

GetMem(), 168
globális deklaráció, 113
globális elérhetőség, 113
goto, 58

## *H*

halmazkezelő eljárások
  exclude(), 95
  include(), 95
halmazkonstansok, 94
halmazműveletek, 96
halmazok
  egyesítése, 95
  különbsége, 95
  metszete, 95
halmaztípusok, 93
high, 19

## *I,Í*

if, 59
implementation, 150, 151
inc, 20
inheritance, 205
inherited, 224
initialization, 150, 152
int(), 44
int64, 19
integer, 17
interface, 102, 221
interfésztípus, 231
IObject, 231
is, 230
IUnknown, 231

## *K*

karakter típusok, 16
karakteres adatok, 18
karaktersorozatok, 22
kivétel, 234
kivételkezelés, 78
kivételkezelő, 234
konstansparaméter, 104
konstruktor, 199
következő, 19

## *L*

label, 58
Lazarus, 2

Length(), 23
library, 149, 159, 221
linefeed, 27
logikai típusú változók, 18
lokális deklaráció, 113
longint, 17
low, 19

## M

maszkolás, 49
matematikai függvények, 42
math modul, 43
MAXINT, 17
MAXLONGINT, 17
megjegyzés, 8
method, 101
metódusok, 193
mod, 40
mutató, 165
mutatótípus, 16
műveletek
  egyoperandusú, 37
  kétoperandusú, 37
műveleti jelek, 37

## N

New(), 166, 197
nil, 166, 219, 222
normál konstansok, 25
not, 47

## O,Ó

object, 193, 218, 223
Object Pascal, 1
object típus, 101
Object-Oriented Programming, 193
objektumok zártsága, 207
objektum-orientált programozás, 101
objektumpéldány, 195
ObjFPC, 215
on...do, 237
OOP, 193
operandusok, 37
operátorok túlterhelése, 119
or, 47
ord, 19
osztálymetódusok, 219
out, 104
overloading, 116, 199
override, 226

## Ö,Ő

önhivatkozó adatstruktúrák, 174
öröklés, 193, 205
ős, 205
összetett utasítás, 59

## P

packed, 79
paraméterek alapértéke, 118
pi, 42
pointer, 165, 168
polymorphism, 210
precedencia, 39
preceding, 19
pred, 19
private, 202, 220
procedure, 103
process task, 243
program, 7
programfej, 7
programtörzs, 8
property, 101, 215
protected, 202, 220, 248
public, 202, 220
published, 218, 220, 221

## Q

qword, 19

## R

raise, 78, 235
random(), 42
read, 27, 31
Read(), 254
readln, 9, 10, 27, 31

real, 17
reAllocMem(), 168
redefine, 206
rekordelemű tömbök, 89
rekordtípusok, 85
rekordtípusú konstansok, 88
rekordtípusú paraméter, 134
rekurzió, 142
rekurzív algoritmusok, 116
rekurzív alprogram, 142
repeat, 59
resourcestring deklaráció, 235
result, 105, 120
résztartomány típus, 22
round(), 44
RTTI, 220
run-time library, 234
runtime type information, 220

## S

Seek(), 254
self, 195, 219
set, 93
SetLength(), 23, 172
shl, 50
ShortString, 22
shr, 50
sizeof, 16
sokalakúság, 193, 210
sorszámozott típusok, 19
sqr(), 42
standard output, 27
statikus élettartam, 115
statikus metódusok, 210
str(), 44
string, 13, 22, 122
Strings unit, 24
StrToFloat(), 44
strukturált típusok, 16
  array, 79
  class, 79
  file, 79
  object, 79
  record, 79
  set, 79
strukturált utasítások, 59
succ, 19
succeeding, 19
System unit, 12, 24, 42, 152, 219
SysUtils unit, 24, 44, 234, 235, 240

## Sz

szabványos kimenet, 27
számok, 13
származtatott, 205
szövegkonstans, 13, 23
sztingműveletek, 52

## T

tartalmazás-vizsgálat, 96
TextToFloat(), 44
threadvar, 245
TInterfacedObject, 231
típus nélküli paraméter, 140
típus-átalakítás, 43
típusos konstansok, 25
TObject, 219
többkiterjedésű tömbök, 80
tömbtípusok, 79
tömbtípusú paraméter, 127
TP mód, 11
true, 18
trunc(), 44
try...except, 236, 238
try...except...end, 78
try...finally, 239
try...finally...end, 78
TThread, 247, 250
tulajdonságok, 101
túlterhelés, 116, 199
Turbo Pascal 7, 1, 4

## Ty

type, 24, 166

## U,Ú

újradefiniálás, 206
unary operátor, 37
Unicode, 11
unit, 149, 202, 221
uses, 151

utasítás, 12

## *Ü,Ű*

üres utasítás, 58

## *V*

val(), 44
valós adatok, 17
változók, 15
var, 15
variálható rekord, 87
variáns típus, 16
virtual, 211, 226
virtuális metódustábla (*VMT*), 211
vituális metódusok, 211

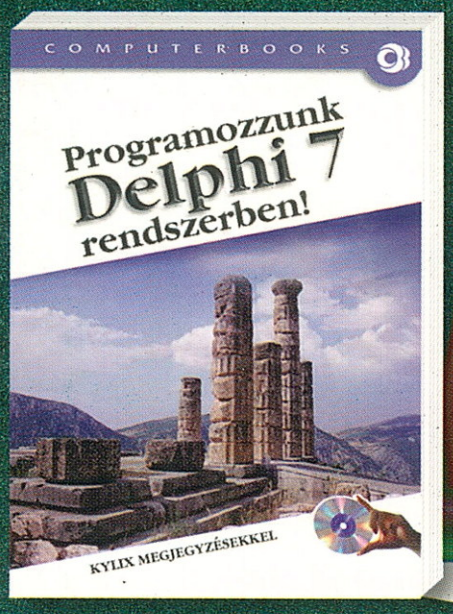
## *W*

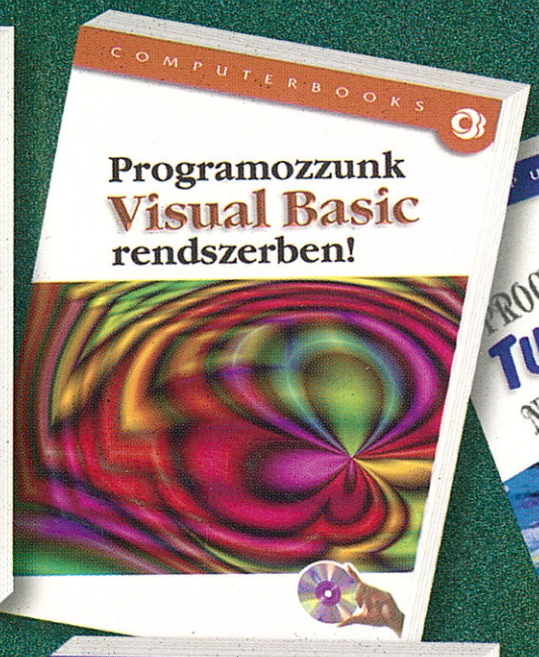
while, 59
WideChar, 18
with, 59, 78, 197, 222
write, 9, 10
writeln, 9

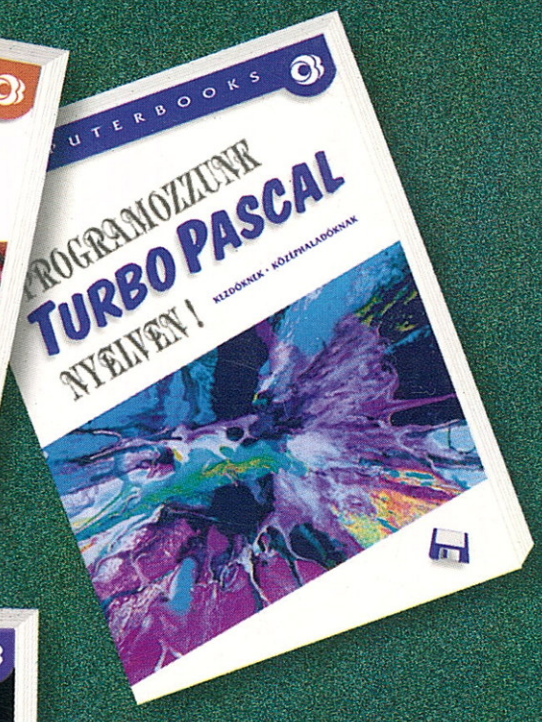
## *X*

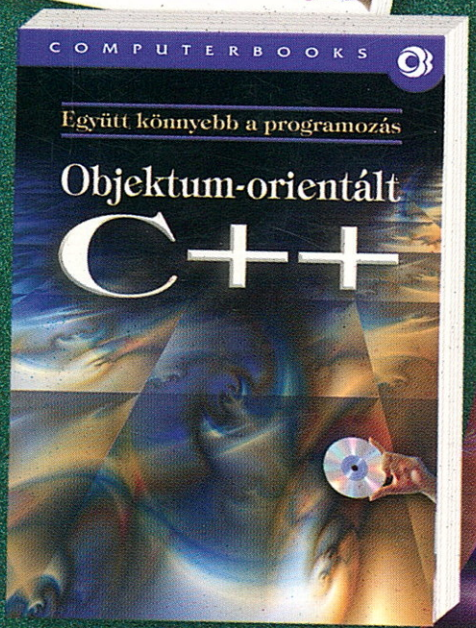
xor, 47

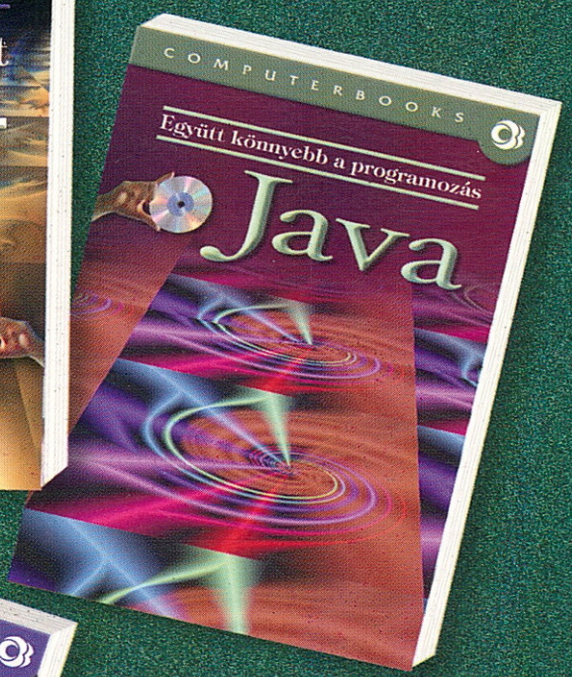
# PROGRAMOZZUNK... sorozatunk

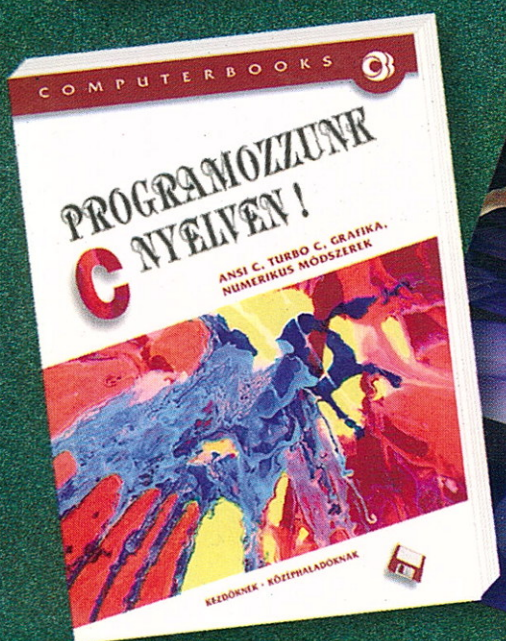


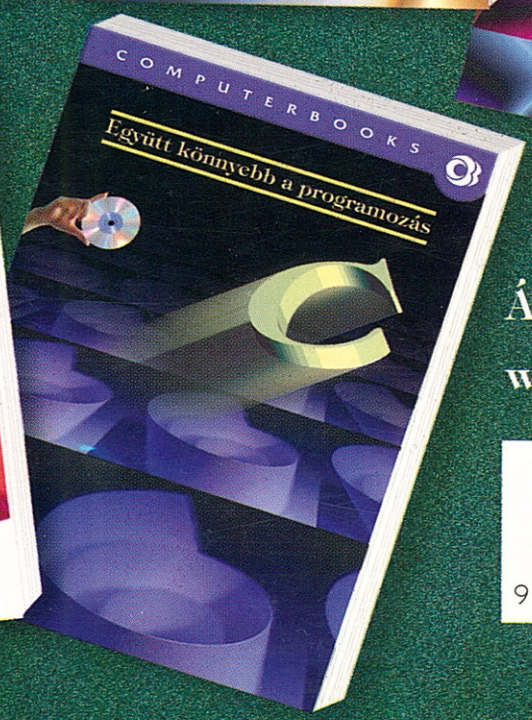






**Ára: 3300,–Ft**

**www.computerbooks.hu**

ISBN 963618348-1